AXIOMS AND THEOREMS FOR INTEGERS, LISTS AND FINITE SETS IN LOGIC FOR COMPUTABLE FUNCTIONS (LCF)

Malcolm Newey

Stanford University

Prepared for:

Office of the Secretary of Defense
Advanced Research Projects Agency
National Aeronautics and Space Administration

January 1973

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-184

STAN-CS-73-330

AXIOMS AND THEOREMS FOR INTEGERS, LISTS AND FINITE SETS IN LCF

BY

MALCOLM NEWEY

SUPPORTED BY
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

JANUARY 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Stanford University, Computer Science Department
Stanford California 94305

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

Axioms and Theorems for Integers, Lists and Finite Sets in LCF

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

technical, January 1973

5. AUTHOR(S) *(First name, middle initial, last name)*

Malcolm Newey

6. REPORT DATE

January 1973

7a. TOTAL NO. OF PAGES

~~53~~ 57

7b. NO. OF REFS

4

8a. CONTRACT OR GRANT NO.

SD-183

b. PROJECT NO.

ARPA Order No. 457

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

STAN-CS-73-330
AIM184

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

Distribution Unlimited

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

LCF (Logic for Computable Functions) is being promoted as a formal language suitable for the discussion of various problems in the Mathematical Theory of Computation (MTC) (TC). To this end, several examples of MTC problems have been formalised and proofs have been exhibited using the LCF proof-checker. However, in these examples, there has been a certain amount of ad-hoc-ery in the proofs: namely, many mathematical theorems have been assumed without proof and no axiomatisation of the mathematical domains involved was given. This paper describes a suitable mathematical environment for future LCF experiments and its axiomatic basis. The environment developed, deemed appropriate for such experiments, consists of a large body of theorems from the areas of integer arithmetic, list manipulation and finite set theory.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

iii

Unclassified

Security Classification

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-184

COMPUTER SCIENCE DEPARTMENT
REPORT CS-330

JANUARY 1973

Axioms and Theorems for Integers, Lists and Finite Sets in LCF.

by

Malcolm Newey

ABSTRACT:

LCF (Logic for Computable Functions) is being promoted as a formal language suitable for the discussion of various problems in the Mathematical Theory of Computation (MTC). To this end, several examples of MTC problems have been formalised and proofs have been exhibited using the LCF proof-checker. However, in these examples, there has been a certain amount of ad-hoc-ery in the proofs; namely, many mathematical theorems have been assumed without proof and no axiomatisation of the mathematical domains involved was given. This paper describes a suitable mathematical environment for future LCF experiments and its axiomatic basis. The environment developed, deemed appropriate for such experiments, consists of a large body of theorems from the areas of integer arithmetic, list manipulation and finite set theory.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defence under Contract SD-183 and in part by the National Aeronautics and Space Administration under Contract NSR 05-020-500.

Axioms and Theorems
for Integers, Lists and Finite Sets
in LCF.

by

Malcolm Newey

CONTENTS

1. INTRODUCTION

By LCF, I mean the Milner version of a logic proposed by Dana Scott in 1969, mechanized by Milner in 1971, and described by Milner in [1,2]. [1] is actually the user's manual for the LCF proof-checker which has been the vehicle for generating formal proofs in the logic.

Since the development of the proof-checker, LCF has been successfully applied to various traditional problem areas of the Mathematical Theory of Computation. The principal experiments have involved program semantics, correctness of programs, termination of programs and compiler correctness [2,3,4].

In each of the examples reported a machine checked proof was generated which increased the reliability of the solution enormously. However, each proof also made a large number of assumptions in the forms of unproved theorems and redundant axioms. Although it can be demonstrated that the particular assumptions involved do not invalidate those experiments, it is clear that the proofs would be considerably more reliable if a solid axiomatic theory was already available to give all the required background results.

The three particular areas of mathematical knowledge which are developed in this paper, namely integer arithmetic, list manipulation and a theory of finite sets, are very important in computation. Moreover, in proving assertions about programs, these theories provide most of the mathematical material which would be classified as background results.

The current project has been to develop a very large theorem bank which will act as an appropriate mathematical environment for future applications of LCF. So far over 300 theorems have been proved (with the aid of the LCF proof-checker, of course) from the axioms given in this paper.

Although there is no distinction possible (in the LCF system) between axioms and definitions (both are declared as AXIOMs), effort was made in the axiomatisation to introduce new functions as terms of the logic. This strategy makes it easier to demonstrate consistency for the sets of axioms presented. Similarly, in the presentation of AXIOMs a contrast is effected by labelling them either axioms (AX) or definitions (DEF).

The large body of theorems, alluded to above, is organised as a sequence of appendices. All the theorems of any appendix depend on the same group of axioms or definitions and appear in an order which is appropriate for efficient proof of the whole group (by making use of the theorem-using facility of LCF). Note that the indentation of theorems is only to make the page layout a little prettier.

2. THEOREMS FROM NO AXIOMS AND A PROPOSITIONAL LOGIC

Appendix 1 gives a number of theorems that require no axioms (strictly - no nonlogical axioms) for their proof in LCF. All can be proved in a few lines but it shortens and so helps to clarify later proofs if they are available.

The theorems

$$\forall p.\, p \rightarrow TT, FF \equiv p$$
$$\forall p.\, p \rightarrow UU, UU \equiv UU$$
$$[\lambda x.\, UU] \equiv UU$$

are important as permanent members of the simplification set of the LCF proof checker. It is also worth mentioning that the block of results exemplified by

$$p \rightarrow TT, UU \equiv FF \vdash TT \equiv FF$$

are designed to make use of the proof by contradiction facility in LCF which 'knows' that TT≡FF (and a few similar wffs) is a contradiction.

A function from and to the domain of truth values which represents the logical NOT operation is readily defined in LCF as

**DEF 2.1 $\quad \neg \equiv [\lambda x.\, x \rightarrow FF, TT]$.

Appendix 2 shows that it behaves according to the truth table

x	¬x
TT	FF
FF	TT
UU	UU

Unfortunately there is no such definition possible to give a suitable meaning to the logical AND or the logical OR operators. The truth table we would like for OR, say, is given as

		y = TT	y = FF	y = UU
x	TT	TT	TT	TT
x	FF	TT	FF	UU
x	UU	TT	UU	UU

We therefore axiomatize the relation as below and note that each axiom is trivially faithful to the above truth table. Moreover the theorems of Appendix 2 show the whole truth table is derivable.

***AX 2.2 $\forall P.\ P \vee TT \equiv TT$
***AX 2.3 $\forall P.\ P \vee FF \equiv P$
***AX 2.4 $\forall P.\ P \vee UU \equiv (P \rightarrow TT, UU)$

An appropriate definition for logical AND is now possible (see below) in terms of the OR operation. We also give an explicit definition of equivalence. The results of appendix 2 give the truth tables for these operators shown below.

**DEF 2.5 $\wedge \equiv [\lambda x\ y.\neg((\neg x) \vee (\neg y))]$
**DEF 2.6 $= \equiv [\lambda x\ y.x \rightarrow y, (y \rightarrow FF, TT)]$

x ∧ y	y = TT	y = FF	y = UU
x = TT	TT	FF	UU
x = FF	FF	FF	FF
x = UU	UU	FF	UU

x = y	y = TT	y = FF	y = UU
x = TT	TT	FF	UU
x = FF	FF	TT	UU
x = UU	UU	UU	UU

3. INDIVIDUAL EQUALITY AND DEFINEDNESS

In the domain of individuals of the logic, we want (very often in practice) to utter sentences which contain terms such as 'x is the same as y'. For example we could require a function

$$f \equiv [\lambda x.\ (\text{is-the-same-as}(x,a) \rightarrow b, g(x))]$$

or we might want a sentence such as

$$\neg(\text{is-the-same-as}(x,y)) :: g(x,y) \equiv h(x,y) \quad .$$

The '$\equiv$' connective of LCF is the most obvious candidate but it cannot be represented by an LCF term since it is not monotonic. What we want is a two place predicate '=' which

i) is undefined exactly when one (or both) of its arguments is undefined,

and otherwise

ii) has the value TT if and only if the two arguments are the same element (not UU).

Such a predicate, obviously monotonic, is possible with appropriate domains of individuals (see below) but as with the logical operators AND and OR, this 'computable' equality cannot be defined but must be axiomatised. The following capture the desired predicate:

***AX 3.1 $\quad \forall x.\ ((x=x) \rightarrow x, UU) \equiv x$

***AX 3.2 $\quad \forall x\ y.\ (x=y) :: x \equiv y$

***AX 3.3 $\quad \forall x\ y.\ (x=x) \rightarrow ((y=y) \rightarrow TT, UU), UU \equiv (x=y) \rightarrow TT, TT$

***AX 3.4 $\quad (UU=UU) \equiv UU$

First note that this equality predicate for the domain of individuals and the logical equivalence predicate defined in the last section are of different types (in the technical sense) and are only given the same name because of shortage of symbols. As with the symbol UU (which denotes an individual, a truth value and an infinite number of functions of different types) the particular predicate intended by '=' can be determined by context.

The role that the first three axioms play is quite straightforward:-

3.1 says that the '=' relation is reflexive on all individuals except UU; It says nothing about UU=UU;

3.2 says that the relation is only true in the reflexive case;

3.3 interpreted in the light of 3.4, this axiom gives us that if neither x,y are UU then x=y is either TT or FF; It also gives that if x=y is TT or FF then neither x or y is the undefined element.

The axiom 3.4 is not really necessary in that if there is any element in the domain of individuals (distinguishable from UU) then 3.4 follows from 3.1-3.3 . For, supposing X to be distinguishable from UU , X≡UU is a contradiction and so we argue by cases on UU=UU : If UU=UU≡TT then X=UU≡TT by monotonicity and X≡UU by axiom 3.2 ; If UU=UU≡FF then X=X≡FF by monotonicity and X≡UU by axiom 3.1 ; Since the TT and FF cases lead to contradictions we have UU=UU≡UU.

Although we are indeed only interested in nontrivial domains we want to be able to prove a body of useful theorems about equality without mentioning any particular elements. 3.4 is needed to prove several of the theorems of appendix 3 and this forces us to add it. For example, the theorem

$$\forall X.\ X=UU \equiv UU$$

can not follow from the first three axioms since in the trivial domain of just UU, we can have UU=UU≡TT and the axioms are satisfied.

X≡Y can always be deduced from X=Y≡TT as prescribed by the axioms, but we also easily get theorems for going the other way

$$X\equiv Y,\ X=X\equiv TT \vdash X=Y\equiv TT$$
$$X\equiv Y,\ Y=Y\equiv TT \vdash X=Y\equiv TT$$

and 2 versions of the commutative law for '=' .

$$\forall X\ Y.\ X=Y \equiv Y=X$$
$$X=Y\equiv TV \vdash Y=X\equiv TV$$

The fact that every element except UU is equal (≡) to itself, gives us the definedness predicate for individuals by definition.

**DEF 3.5 $\partial \equiv [\lambda x.\ x=x]$

where ∂ will be TT on all individuals except UU and ∂(UU) will be UU.

Appendix 3 also gives useful theorems about the ∂ predicate. Note especially the following theorems which are extremely important when arguing by cases on the definedness of some individual:-

$$\partial(X)\equiv FF \vdash TT\equiv FF \qquad\qquad \partial(X)\equiv UU \vdash X\equiv UU .$$

It was inferred above, that the axioms for '=' dictate some structure for the domain of individuals. This structure is simply flatness or discreteness (which means that for any element X, if Y⊂X then Y is either UU or X itself). The following theorems show that this is so and it is asserted that flatness isn't a high price to pay for the notions of equality and definedness. In fact, Scott, in his original proposal suggested that this was a reasonable assumption.

$$X=Y\equiv FF,\ X\subset Y \vdash TT\equiv FF$$
$$\partial(X)\equiv TT,\ X\subset Y \vdash X\equiv Y$$

4. NATURAL NUMBERS

The natural numbers can be axiomatized by the following four axioms and four definitions:

```
**DEF 4.1      Z ≡ [λx.x=0]
***AX 4.2      Z(0) ≡ TT
**DEF 4.3      isnat ≡ [αF.[λx.Z(x)→TT,F(pred(x))]]
***AX 4.4      ∀X.isnat(X)::Z(X)→0,succ(pred(X)) ≡ X
***AX 4.5      ∀X.isnat(X)::Z(succ(X)) ≡ FF
***AX 4.6      ∀X.isnat(X)::pred(succ(X)) ≡ X
**DEF 4.7      1 ≡ succ(0)
**DEF 4.8      2 ≡ succ(1)
```

where the axiomatised quantities are the individual `0', the function `succ' and the function `pred'.

A glance at appendix 4 shows that many ususal properties of the natural numbers are provable. In particular, the following ones:-

```
isnat(0) ≡ TT
isnat(X)≡TT  ⊢  Z(succ(x))≡FF
isnat(X)≡TT  ⊢  isnat(succ(x))≡TT
isnat(X)≡TT, isnat(Y)≡TT, succ(X)≡succ(Y)  ⊢ X≡Y
g(0)≡TT,  ∀x.isnat(x)::g(x)::g(succ(x))≡TT  ⊢  ∀x.isnat(x)::g(x)≡TT
```

which approximate PEANO Axioms for natural numbers. I use the word `approximate' since the free variable `g' in the induction theorem can only be instantiated to a continuous function. However, because domain of individuals we use is discrete, if F is any function on just the natural numbers, it can be extended to a continuous function by defining F(UU) to be UU. Hence theorems which follow from the Peano postulates in usual logics will be valid (perhaps with relativisation) in this LCF environment.

See also appendix 5 where a proof of the induction theorem is given as an example of a technique of using Scott induction to prove relativised assertions. It should also be noted that this induction theorem can be applied to prove assertions of the form

```
∀x.isnat(x)::h(x)≡k(x)
```

by instantiating g with the term [λx.h(x)=k(x)] and proving

```
h(0)=k(0)≡TT, ∀x. isnat(x):: h(x)=k(x):: h(succ(x))=k(succ(x))≡TT .
```

Note that this doesn't mean that the following sentence is a theorem:

```
h(0)≡k(0), ∀x.isnat(x)::h(x)=k(x)::h(succ(x))≡k(succ(x))
      ⊢  ∀x.isnat(x)::h(x)≡k(x)
```

for consider the functions h ≡ [λx.UU] and k ≡ [λx.Z(x)→UU,0].

Similarly, the instantiation g←[λx.h(x)←FF,TT] means that the theorem can be applied to attack goals of the form

$$\forall x.\ \mathrm{isnat}(x) :: h(x) \equiv FF$$

We would now like to argue (informally) that there are no non-standard models satisfying the axioms. We already have that $\mathrm{succ}^n(0)$ behaves as the integer n so we need only prove that the set $\{\mathrm{succ}^n(0)\}$ exhausts the set of things for which 'isnat' is true.
Reasoning outside LCF we can say

$\mathrm{pred}(x) \equiv y, \mathrm{isnat}(y) \equiv TT, \mathrm{isnat}(x) \equiv TT \vdash x \equiv \mathrm{succ}(y)$ is provable;

Hence, for any integer n,

$\mathrm{pred}^n(X) \equiv 0, \mathrm{isnat}(X) \equiv TT \vdash X \equiv \mathrm{succ}^n(0)$ is provable;

But we know from the recursive definition of isnat

if $\mathrm{isnat}(X) \equiv TT$ then $\mathrm{pred}^n(X) \equiv 0$ for some n;

So $\mathrm{isnat}(X)$ implies $X \equiv \mathrm{succ}^n(0)$ for some n.

It is clear from the various preceding comments that the set of axioms given is consistent and a faithful representation of the natural numbers. We now consider redundancy in the axioms and note

4.2 is terse and basic; Without it is is not possible to derive $\mathrm{isnat}(0) \equiv TT$ or even that there exist any natural numbers;

4.4 may not be condensed to $\forall x.\ Z(x) \rightarrow 0, \mathrm{succ}(\mathrm{pred}(x)) \equiv x$ as there may be elements in the domain of individuals on which 'pred' is undefined and so (noting that $\mathrm{succ}(UU) \equiv UU$ will be derivable) we get a condradiction.
4.4 cannot be weakened to either of the sentences
$\forall x.\ \mathrm{succ}(\mathrm{pred}(x)) \equiv x$; $\forall x.\ \mathrm{isnat}(x) :: \mathrm{succ}(\mathrm{pred}(x)) \equiv x$
without making a commitment to the existence of an element given by pred(0). If the axioms are to be used as a base for the integers this is OK but if the only numbers are to be the natural numbers then we would want $\mathrm{pred}(0) \equiv UU$ to be true.

4.5 is needed to get the distinctness of $\mathrm{succ}^m(0)$ and $\mathrm{succ}^n(0)$; Without the axiom at all, it is not possible to show that 0 and 1 are not the same element. With only $Z(1) \equiv FF$ in its place, it cannot even be reasoned that 0 and succ(succ(0)) are distinct;

4.8 is a basic property which cannot be derived from the other axioms.

It should be noted that the functions 'succ' and 'pred' are only partially specified in the natural number axioms since we want them to be defined appropriately when we axiomatize the set of integers (both positive and negative).

Care has been taken in assembling the appendix of theorems to exhibit the role that equality plays in the axiomatisation. The first group of theorems depends only on axioms 4.2 to 4.8 which do not mention equality or definedness. The later theorems require the equality axioms and 4.1 as well for their demonstration.

5. INTEGERS AND ARITHMETIC

***AX 5.1 $\forall x.\ isnat(x) :: pos(x) \equiv Z(x) \rightarrow FF, TT$

***AX 5.2 $\forall x.\ pos(x) :: isnat(x) \equiv TT$

***AX 5.3 $\forall x.\ pos(mns(x)) \equiv pos(x) \rightarrow FF, Z(x) \rightarrow FF, TT$

***AX 5.4 $\forall x.\ pos(x) \rightarrow TT, TT \equiv isint(x) \rightarrow TT, UU$

***AX 5.5 $\forall x.\ isint(x) \rightarrow mns(mns(x)), mns(x) \equiv isint(x) \rightarrow x, UU$

***AX 5.6 $\forall x.\ succ(x) \equiv mns(pred(mns(x)))$

***AX 5.7 $\forall x.\ pred(x) \equiv mns(succ(mns(x)))$

***AX 5.8 $[\lambda x.\ isint(x) \rightarrow TT, TT] \equiv \partial$

The interpretation intended here is that a positive integer `n' , say, is represented by $succ^n(0)$ and that a negative integer `-m', say, is represented by $pred^m(0)$. Obviously `mns' is the unary minus operator and `pos' is the greater-than-zero predicate. Appendix six gives a large collection of basic, but useful, theorems provable from the axioms of sections 3,4,5. Note that the functions `isnat', `pos', `mns', `succ' and `pred' are all undefined where `isint' isn't true.

Just about all that will be claimed about the above axioms for integers in LCF is that they are consistent (since each is true in the standard intepretation of the integers) and the usual theorems can be proved using them. Because they are just a bunch of suitable properties which together do the job, no individual deserves comment.

It is readily demonstrated that $\{ succ^n(0) \} \cup \{ pred^m(0) \}$ is the same set as $\{ x \mid isint(x) \equiv TT \}$ as follows:
Suppose $isint(X) \equiv TT$;
From AX5.4 we get that $pos(X)$ must be TT or FF:
If $pos(X) \equiv TT$ then $isnat(X) \equiv TT$ and so $X \equiv succ^n(0)$ for some $n>0$;
If $pos(X) \equiv FF$ then $isnat(mns(X)) \equiv TT$ and so $mns(X) \equiv succ^n(0)$ for some $n \geq 0$ giving $X \equiv mns(succ^n(0))$;
But $[\lambda x.mns(succ(x))] \equiv [\lambda x.pred(mns(x))]$ so we get $X \equiv pred^n(0)$;
Hence $isint(X) \equiv TT$ implies $X \equiv succ^n(0) \vee X \equiv pred^n(0)$ for some $n>0$.
Also we see that $isint(succ^m(0)) \equiv TT$ for all $m \geq 0$ from the theorem

$$isint(X) \equiv TT \vdash isint(succ(X)) \equiv TT$$

and $isint(pred^m(0)) \equiv TT$ for all $m \geq 0$ from the corresponding theorem

$$isint(X) \equiv TT \vdash isint(pred(X)) \equiv TT .$$

Although none of the theorems of appendix 6 are deep, one can see how many important simple relations there are between the objects axiomatised in this section.

The main induction theorem for integers is simply stated thus:-

$$g(0) \equiv TT, \forall x.\ isint(x) :: g(succ(x)) \equiv g(x) \vdash \forall x.\ isint(x) :: g(x) \equiv TT .$$

To prevent confusion arising from the similarity between this theorem and the induction principle for natural numbers, note the following NON-theorem:-

$$g(0) \equiv TT,\ \forall x.\ isint(x) :: g(x) :: g(succ(x)) \equiv TT \vdash \forall x.\ isint(x) :: g(x) \equiv TT$$

The discussion of the corresponding induction principle for natural numbers introduced a technique which is appropriate, in this section also, for attacking goals of the form $\forall x.h(x)\equiv k(x)$ using such a rule. That was to instantiate the 'g' of the theorem with the term $[\lambda x.h(x)=k(x)]$. Practice shows, however, that it is economical to restate the theorem so as to incorporate the idea :-

$$
\begin{array}{l}
h(0)\equiv k(0),\\
\forall x.\,isint(x)::\partial(h(x))\equiv TT,\\
\forall x.\,isint(x)::\partial(k(x))\equiv TT,\\
\forall x.\,isint(x)::(h(x)=k(x))::h(succ(x))\equiv k(succ(x)),\\
\forall x.\,isint(x)::(h(x)=k(x))::h(pred(x))\equiv k(pred(x)),\\
\vdash\ \forall x.\ isint(x)::\ h(x)\equiv k(x);
\end{array}
$$

Although this is considerably more cumbersome, each notion expressed by the antecedents must be proved any either case and so the economy lies in not having to prove by nested cases arguments

$$\forall x.isint(x)::(h(x)=k(x))\equiv(h(succ(x))=k(succ(x)))$$

With the integers axiomatised satisfactorily, we proceed to definition of the basic arithmetic functions and predicates:-

Functions:

```
**DEF 5.9     + ≡ [αG.[λx y. Z(y)→isint(x)→x,UU,
                        pos(y)→G(succ(x),pred(y)),G(pred(x),succ(y))]]
**DEF 5.10    - ≡ [λx y.x+mns(y)]
**DEF 5.11    * ≡ [αG.[λx y. Z(y)→isint(x)→0,UU,
                       pos(y)→G(x,pred(y))+x, G(x,succ(y))-x]]
**DEF 5.12    / ≡ [αG.[λx y. Z(y)→UU,Z(x)→(isint(y)→0,UU),
                          pos(x)→pos(y)→  pos(y-x)→0,succ(G(x-y,y)),
                                mns(G(x,mns(y))), mns(G(mns(x),y))]]
**DEF 5.13    % ≡ [λx y. x-((x/y)*y)]

**DEF 5.14    Fac ≡ [αG.[λx. Z(x)→1,pos(x)→x*G(x-1),UU]]
**DEF 5.15    Look ≡ [αG.[λx f p. p(x)→x,G(f(x),f,p)]]
```

Predicates:

```
**DEF 5.16    > ≡ [λx y. pos(x-y)]
**DEF 5.17    ≥ ≡ [λx y. Z(x-y)→TT,pos(x-y)]
**DEF 5.18    < ≡ [λx y. y>x]
**DEF 5.19    ≤ ≡ [λx y. y≥x]

**DEF 5.20    even ≡ [λx. Z(x@2)]
**DEF 5.21    odd ≡ [λx. Z(x@2)→FF,TT]
**DEF 5.22    buq ≡ [αG.[λx y p. (x>y)→TT,p(x)→G(x+1,y,p),FF]]
**DEF 5.23    beq ≡ [αG.[λx y p. (x>y)→FF,p(x)→TT,G(x+1,y,p)]]
**DEF 5.24    Pr ≡ [λx. [λy.(y>1)→ buq(2,y-1,[λz.(y@z)=0→FF,TT]),
                                FF](x≥0 →x,mns(x))]
```

Most of these definitions are self explanatory and the others become obvious with a few points of explanation:-

i) '/' is integer division, of course, and '⊗' is the 'mod' operator which gives remainder on division. These are defined in the normal manner for positive integers and are extended (to operations involving negative integers) in such a way that the sign of x/y is always appropriate algebraically and the sign of x⊗y is the same as the sign of x. This choice enables the reconstruction of a number from its quotient and remainder (with respect to a given divisor).

ii) 'Fac' is the factorial function and is only defined for non-negative integer arguments.

iii) Look(x,f,p) yields the first integer y (if any) in the sequence {x, fx, ffx, fffx, ...} which satisfies the predicate p (provided no previous member of the sequence caused p to yield UU).

iv) 'buq' stands for Bounded Universal Quantifer and 'beq' denotes Bounded Existential Quantifier and are meant to take the place of regular quantifiers in numeric proofs. The importance of buq comes from the pair of theorems:

$$buq(X,Y,p)\equiv TT \vdash \forall z.\, z\geq X :: Y\geq z :: p(z)\equiv TT$$
$$\forall z.\, z\geq X :: Y\geq z :: p(z)\equiv TT \vdash buq(X,Y,p)\equiv TT$$

A similar result for 'beq' is expressable as the meta-theorem that (Provided p is total on the range <X,Y>) beq(X,Y,p)≡TT IFF ∃ integer in <X,Y> that satisfies p.

The totality proviso in this result is essential, for if p(n)≡UU and p(n+1)≡TT then beq(n,n+1,p)≡UU even though there does exist an integer in the range which satisfies the given predicate.
Although the predicate which gives TT exactly when there is an appropriate element in the range is definable as
[αG.[λx y p.x>y→TT,p(x)vG(x+1,y,p)]],
DEF 5.23 is preferred because of the useful relationship between that version of beq and the Look function.

v) Pr(x) is TT if either x or mns(x) is a natural number which is prime in the usual sense (not 1). Pr is a total predicate over the integers.

vi) Note that all the functions and predicates take at least one argument which is of type 'individual'. All these functions (except Look) become undefined when applied to individuals which are not integers.

Appendix 7 contains a rather large collection of results that follow from the results on integers and the definitions listed above. There are basic theorems about all of the functions and predicates except $<$ and $\leq$. If a problem contains these predicates then the definitions 5.18 and 5.19 should be applied to transform the goals to ones containing $>$ and $\geq$.

We have already introduced 2 mathematical induction theorems which require, for their application, steps of the forms:-

$$g(x) \vdash g(succ(x)) \qquad g(x) \vdash g(pred(x))$$

Such statements are often as inconvenient to prepare as the result we wish to establish. Actually, we want to model, in LCF, that form of mathematical induction given (in predicate calculus) by:-

$$\{\forall x.\ (\ \forall y.\ [y<x \wedge y\geq 0] \supset p(y)\ ) \supset p(x)\ \} \supset\ [\forall x.\ x\geq 0 \supset p(x)]$$

The obvious problem about what to do with this in LCF, is what to do with the nested quantifiers. Fortunately, the nested quantifier is bounded and so we get the LCF version of the theorem as:-

$$\forall x.\ x\geq 0::\ buq(0,x-1,P)::P(x)\equiv TT \ \vdash\ \forall x.\ x\geq 0::\ P(x)\equiv TT$$

Actually a more primitive form of the theorem was needed to prove certain results about division which preceded the work on relations and 'buq'.

Two more functions which will be similarly treated are the sum and product of a finite sequence - the big SIGMA and big PI notation of analysis.

```
**DEF 5.25      Sum ≡ [αG.[λx y f. y<x → 0, f(x)+G(x+1,y,f)]]
**DEF 5.26      Prod ≡ [αG.[λx y f. y<x → 1, f(x)*G(x+1,y,f)]]
```

6. LISTS and S-EXPRESSIONS

Since lists are a special case of S-expressions, we proceed with an axiomatisation of the more general object.

***AX 6.1	issexp(UU) ≡ UU
***AX 6.2	issexp(NIL) ≡ TT
**DEF 6.3	null ≡ [λx. x=NIL]
**DEF 6.4	atom ≡ [λx. issexp(x)→null(x),TT]
***AX 6.5	∀X. atom(X):: head(X)≡UU
***AX 6.6	∀X. atom(X):: tail(X)≡UU
***AX 6.7	∀X Y. head(cons(X,Y))≡∂(Y)→X,UU
***AX 6.8	∀X Y. tail(cons(X,Y))≡∂(X)→Y,UU
***AX 6.9	∀X. cons(head(X),tail(X))≡atom(X)→UU,X
***AX 6.10	∂ ≡ [αG. [λx. atom(x)→TT,G(head(x))→G(tail(x)),UU]]

Note first that AX 6.1 is valid for all domains which have defined individuals other than S-expressions - the most common circumstance. In situations where all individuals are S-expressions it would be consistent to say that issexp(UU)≡TT but it would be unlikely to give any advantage over postulating issexp(UU)≡UU. Hence, for the sake of proving some handy theorems about S-expressions (which must be true whenever NIL is not the only atom) we assert 6.1 instead of leaving issexp(UU) unspecified.

The purpose of axiom 6.10 is to eliminate (from models) any structures which are infinite. This also means that circularity (which is possible in LISP, for example) is ruled out. As an illustration of the implications of this axiom, a theorem is proved in appendix 8 which gives that if head(X)≡X then X≡UU. A more complete result about circularity is discussed below using the notion of subexpression.

There is one other debatable point about these axioms. It is that we have, as you may have anticipated from the earlier discussion of equality between individuals, adopted the doctrine of discreteness for the domain of S-expressions. The opposing point of view is that a term such as cons(UU,X) (which clearly must be 'under' both the terms cons(A,X) and cons(B,X) for any individuals A & B) is not the same as UU and, moreover, tail(cons(UU,X))≡X. As far as the relative powers of the opposing systems are concerned, it seems that most theorems are identical, but there are a some notions expressable more simply in one system than the other. The big argument in favor of the above set of axioms is that with discreteness comes the notion of equality as expounded earlier. The only tricky part about amending the above axioms to allow for the case where cons(UU,X)≠UU is the problem of excluding the infinite S-expressions.

Appendix 8 contains theorems about the functions issexp, head, tail, cons, atom and null. We mention here only an induction theorem for S-expressions:-

$$\forall x\ y.\ g(x) :: g(y) :: g(cons(x,y)) \equiv TT,\quad \forall x\ y.\ atom(x) :: g(x) \equiv TT \vdash \forall x.\ \partial(x) :: g(x) \equiv TT$$

Following LISP, a list is a special case of an S-expression, namely one which transforms to NIL after some number of applications of the tail operator. As such, lists are easily defined.

```
**DEF 6.11      islist ≡ [αG.[λx. null(x)→TT,atom(x)→FF,G(tail(x))]]
```

As usual, a number of theorems form an appendix (9) but we give an induction theorem locally.

$$\forall x\ y.\ \partial(x) :: islist(y) :: g(y) :: g(cons(x,y)) \equiv TT,\ g(NIL) \equiv TT \vdash \forall x.\ islist(x) :: g(x) \equiv TT$$

A number of usual operations on lists and S-expressions are given with some others that foreshadow the treatment of sets in the next section of this report.

```
**DEF 6.12    rev ≡ [λX.rev2(X,NIL)]
**DEF 6.13   rev2 ≡ [αG.[λx y. null(x)→y,G(tail(x),cons(head(x),y))]]
**DEF 6.14      & ≡ [αG.[λx y. null(x)→y,cons(head(x),G(tail(x),y))]]
**DEF 6.15 ANDmap≡ [αG.[λx p. islist(x)→
                        (null(x)→TT,p(head(x))→G(tail(x),p),FF),UU]]
**DEF 6.16  ORmap ≡ [αG.[λx p. islist(x)→
                        (null(x)→FF,p(head(x))→TT,G(tail(x),p)),UU]]
**DEF 6.17  FNmap ≡ [αG.[λx f.
                       (null(x)→NIL,consf(head(x)),G(tail(x),f))]]
**DEF 6.18  PRUNE ≡ [αG.[λx p.null(x)→NIL,p(head(x))→G(tail(x),p),
                                    cons(head(x),G(tail(x),p))]]
**DEF 6.19    mem ≡ [λx y. ∂(x)→ORmap(y,[λz.x=z]),UU]
**DEF 6.20   memL ≡ [λx y. islist(y)→ANDmap(x,[λz.mem(z,y)]),UU]
**DEF 6.21 memEQ ≡ [λx y. memL(x,y)→memL(y,x),FF]
**DEF 6.22   memS ≡ [λx y. PRUNE(x,[λz.y=z])]
**DEF 6.23  memSL ≡ [λx y.PRUNE(x,[λz.mem(z,y)])]
**DEF 6.24 subexp≡ [αG.[λx y. (x=y)→TT, atom(y)→FF, G(x,head(y))→TT,
                                                    G(x,tail(y))]]
**DEF 6.25  assoc ≡ [αG. [λx y. ∂(x)→ islist(y)→ null(y)→NIL,
                      x=head(head(y))→head(y),G(x,tail(y)),  UU,  UU]]
**DEF 6.26   forL ≡ [αG.[λL f fNIL. null(L)→fNIL,
                                    f(head(L),G(tail(L),f,fNIL))]]
**DEF 6.27  nodes ≡ [αG.[λX.atom(X)→0,succ(G(head(X))+G(tail(X)))]]
**DEF 6.28  length≡ [αG.[λX.null(X)→0,succ(G(tail(X)))]]
```

The function 'rev' is the function which produces a list which is the reverse of the argument list and is defined in the traditional way (using an auxiliary function 'rev2'). '&', the append function is defined as the fixpoint of the appropriate computation. It is proved (see appendix 10) that '&' could have been defined by :

$$\& \equiv [\lambda x\ y.\ rev2(rev(x),y)].$$

Various basic properties of these two important functions are to be found in appendix 10. Note that the second argument of '&' need not be a list for the function to be defined. However, the following result is readily proved (and a similar remark applies to 'rev2'):

$$\forall X.islist(X)::\ islist(X\&Y)\equiv islist(Y)$$

The predicate ANDmap is used to describe situations in which all the elements of a list satisfy some predicate. The computation is performed by applying the predicate to each list element in turn until the end of the list is reached (and the result is TT) or until an element is encountered which does not satisfy the predicate. This method of computation means that, for example, ANDmap(X,p) may be undefined because $p(y)\equiv UU$ for some object y. Because of this fact, many of the basic theorems about ANDmap are based on the assumption that the predicate is total. The predicate ORmap is the disjunctive analogue of ANDmap. The motivation for developing these predicates was to aid in the development of some of the later list operations. There are many theorems proved (see appendix 10) which describe the the interaction between these two maps and 'rev' (or '&').

FNmap is simply a function on lists which applies a function to each member of the argument list. PRUNE is a function, also just defined for lists, which removes from the arument list those elements which satisfy some predicate. As examples, FNmap(X,[λy.y*2]) would double every element of a (numeric) list X and PRUNE(Y,[λx.x<0]) would remove every negative element from a (numeric) list Y.

The group of operations 6.19 to 6.23 are concerned with membership in lists and are crucial to the theory of sets given in the next section. mem(x,L) will be true whenever x is one of the elements of list L. It is shown in the theorems that the following is an alternate definition of 'mem':-

$$mem \equiv [\alpha G.[\lambda x\ y.\ islist(y)\rightarrow null(y)\rightarrow \partial(x)\rightarrow FF, UU\ (x=head(y))\rightarrow TT, G(x,tail(y)),\ UU]].$$

memL(X,Y) will be TT whenever ALL the elements of list X are members of list Y also. The following is an alternate definition for 'memL':-

$$memL \equiv [\alpha G.[\lambda x\ y.\ islist(y)\rightarrow\ islist(x)\rightarrow null(x)\rightarrow TT, mem(head(x),y)\rightarrow G(tail(x),y), FF,\ UU,\ UU]].$$

memEQ(X,Y) simply indicates whether two lists, X and Y, have the same elements (independent of the order or multiplicity of those elements). memS(L,X) deletes all elements of list L which are occurrences of the object X while memSL(L,M)

deletes all elements of list L which are also elements of list M.

The function `subexp' is principally used to indicate the imbedding of one S-expression in another. subexp(X,Y) is TT exactly when some sequence (possibly null) of head and tail operations take object Y into object X. Thus if Y is an S-expression then subexp(X,Y) indicates that X is imbedded in Y (at least once) but if Y is an atom then subexp(X,Y) indicates that X is the same atom. We are now able, using this new notion, to prove in LCF the non-existence of certain infinite S-expressions.

```
subexp(X,Y):: subexp(Y,X):: X≡Y
```

The infinite lists forbidden by this theorem are the ones which in LISP could be represented using circularity.

The function `assoc' is purely LISP-inspired and could be useful where some association technique is appropriate to a proof. An alternate way of defining `assoc' would be as:-

```
        assoc ≡ [λx y.lookL(y,[λz.head(z)=x])]
where
        lookL ≡ [αG.[λL p. islist(L)→null(L)→NIL,
                           p(head(L))→head(L),G(tail(L),p),  UU]]
```

is, in general, a more useful function. However, such a function which looked for the first element of a list to satisfy a given predicate could be more suitably defined since with this definition lookL(X,p)≡NIL could mean EITHER p(NIL)≡TT and NIL is a member of X OR that no element of X satisfied P.

The function `forL' is a device for simplifying definitions of other functions which take a list as their only argument and which compute from the tail of the list to the head. As an example, the sum of the elements of a numeric list X is given by forL(X,+,0) while the product is given by forL(X,*,1). One could also give slightly more compact definitions of `PRUNE' and `FNmap' (and predicates which are similar to `ANDmap' and `ORmap') using `forL'.

The function `nodes' counts the subexpressions of an S-exprn. which are not atomic or the number of nodes in a tree representation of the S-exprn. `length' is simply the number of elements in a list and could have been defined (to further illustrate `forL'):-

```
        length ≡ [λx.forL(x,[λy z.z+1],0)].
```

These last two functions (which are the only ones to refer to the notions developed for arithmetic) are not expounded in the appendix but the usual properties clearly follow from the definitions and the arithmetic environment already constructed and described.

7. FINITE SETS

Sets turn out to be quite hard to categorise in LCF, even finite ones. The difficulty arises from the lack of existential quantifiers or the lack of nested quantification, depending how you look at it. The problem occurs even as soon as you try to define the empty set and give its properties. We can easily express that nothing is in this set (call it NS) by the wff ∀x. ∂(x):: x∈NS≡FF but when we come to say that the null set is the ONLY set in which there is nothing, we find no simple way to express the sentence

∀x.x∈A≡FF ⊢ A≡NS as a well-formed formula of LCF.

Recall that the form of an axiom in LCF is a WFF - not a sentence.

The solutions we discovered to the above problem all involved axiomatising a choice function for sets which would pick some element from any set it was applied to. However, using this notion, several developments of the theory are possible. Because of the enormous economy involved, we have based our set theory on transformations between sets and lists. The choice function involved is the taking of the head of the list that a given set maps into (see the function `select' defined below).

The transformation functions are `listof' and `setof' and are axiomatised as follows; note that finiteness is automatic since lists were axiomatised to be finite.

```
***AX 7.1    [λx.isset(x)→TT,TT] ≡ ∂
***AX 7.2    ∀x.isset(setof(x))≡(islist(x)→TT,UU)
***AX 7.3    ∀x.islist(listof(x))≡(isset(x)→TT,UU)
***AX 7.4    ∀x.setof(listof(x))≡(isset(x)→x,UU)
***AX 7.5    ∀x y. memEQ(x,y) ≡ setof(x)=setof(y)
```

Note that these axioms do not imply that sets are disjoint from lists, S-expressions or any other data type that may be part of individuals. In fact it is not inconceivable to identify sets with the lists to which they map by `listof'. However, all that is needed to ensure disjointness is an axiom like

∀x. isset(x):: issexp(x)≡FF

With these notions, we easily DEFINE all the usual operations on sets in terms of the list membership functions and predicates defined in the last section. We start with some basic ones:-

```
**DEF 7.6    NS  ≡ setof(NIL)
**DEF 7.7    ∈   ≡ [λx y. mem(x,listof(y))]
**DEF 7.8    subset ≡ [λx y. memL(listof(x),listof(y))]
**DEF 7.9    ∪   ≡ [λx y.setof(listof(x)&listof(y))]
**DEF 7.10   \   ≡ [λx y.setof(memSL(listof(x),listof(y)))]
**DEF 7.11   ∩   ≡ [λx y.setof(memSL(listof(x),listof(x\y))]
**DEF 7.12   select ≡ [λx. head(listof(x))]
**DEF 7.13   singtn ≡ [λx. setof(cons(x,NIL))]
```

With regard to these definitions, it will suffice to note :-

i) NS is to be taken to be the null (or empty) set;
ii) `ϵ' is the set membership predicate;
iii) XUY denotes the union of the sets X and Y;
iv) X∩Y denotes the intersection of the sets X and Y;
v) `\' is the set subtraction operation;
vi) `select' is the choice function for picking elements from non-empty sets;
vii) singtn(X) denotes the set with X as it's only element.

The definitions just given are the basic set operations for which theorems have been proved in LCF (for this project). Appendix twelve contains theorems relevent to these operations.

There are many theorems displayed in appendix 12 but consider how similar the following short collection of provable results is to the usual predicate calculus axioms for set theory. In fact, it is possible to prove all the other results of appendix 12 (except those that mention the functions `listof' or `setof') just from these theorems. Can, therefore, these sentences be taken as an alternate basis for a set theory in LCF? No! Two of these theorems have universal quantifiers in the assumptions and as noted earlier, only sentences with no assumptions are admissable as axioms. Note another disadvantage: none of the set operations are introduced by explicit definition.

[λx. isset(x)→TT,TT] ≡ ∂

∀X Y. XϵY→TT,TT ≡ ∂(X)→(isset(Y)→TT,UU),UU

isset(Y)≡TT, ∀W. WϵX ≡ WϵY ⊢ X ≡ Y

∂(X) ≡ TT ⊢ XϵNS ≡ FF

∀X Y. subset(X,Y)→TT,TT ≡ isset(X)→(isset(Y)→TT,UU),UU

isset(X)≡TT, isset(Y)≡TT, ∀W. WϵX:: WϵY≡TT ⊢ subset(X,Y)≡TT

subset(X,Y)≡TT ⊢ ∀W. WϵX:: WϵY≡TT

∀W X Y. Wϵ(XUY) ≡ (WϵX)→ isset(Y)→TT,UU, (WϵY)→TT,FF

∀W X Y. Wϵ(X\Y) ≡ (WϵX)→ (WϵY)→FF,TT, isset(Y)→FF,UU

∀W X Y. Wϵ(X∩Y) ≡ (WϵX)→ (WϵY)→TT,FF, isset(Y)→FF,UU

∀W X. Wϵsingtn(X) ≡ ∂(W)→(isset(X)→(W=X),UU),UU

There are some other very important set operations which have been defined appropriately (see below) but (mainly because of lack of time) no rigorous development of their properties has been done.

```
**DEF 7.14      forS≡ [αG.[λS f fNS. (x=NS)→fNS, f(select(x),
                                    G(x\singtn(select(x)),f,fNS) ) ]]
**DEF 7.15      Un  ≡ [λx.forS(x,[λy z.yUz],NS)]
**DEF 7.16      In  ≡ [λx.forS(x,[λy z.yΩz],x)]
**DEF 7.17      reduce≡ [λx p. forS(x,[λy z. p(y)→singtn(y)Uz,z],NS)]
**DEF 7.18      seq ≡ [λx p. (reduce(x,p)=NS)→FF,TT]
**DEF 7.19      suq ≡ [λx p. reduce(x,p)=x ]
**DEF 7.20      PS  ≡ [αG.[λx. forS(x,[λy z.G(x\y)Uz],singtn(x))]]
**DEF 7.21      Card≡ [λx.forS(x,[λy z.z+1],0)]
```

where, in words,

i) forS is just an important auxiliary function;

ii) Un(X) is the n-way union of all the sets that are in X;

iii) In(X) is the n-way intersection of the elements of X;

iv) reduce(X,p) is used to denote the set which in normal notation is written { z | z∈X ∧ p(z) };

v) 'seq' denotes Set Existential Quantifier & seq(X,p)≡TT when there is a member of X which satifies predicate 'p' and 'p' is defined on the rest of the set;

vi) 'suq' denotes Set Universal Quantifier and seq(X,p)≡TT iff predicate 'p' is TT on all elements of set X;

vii) PS is the power set function;

iix) Card is the cardinality function for sets.

8. CONCLUSION

AXIOMATISATION TECHNIQUES.

In this work certain techniques were used in axiomatising various mathematical notions. To illustrate these we take an abstract example: "Axiomatise boops using the previously axiomatised notion of beeps !"

We start working with the assumption that there will be things in the domain of individuals that are not boops , not beeps (which may overlap with the set of boops) and are not anything that is mentioned in the axioms that the 'boop axioms' will depend on. This assumption means that many theorems about boops will have to be relativised but it also guarantees that we will be able combine such groups of axioms without fear of inconsistency. Relativisation is only possible if there is a predicate 'isboop' which will be true only on boops. We will probably want

$$\partial \equiv [\lambda x.\ \mathrm{isboop}(x) \rightarrow TT, TT]$$

to be true and if this is not provable from the other 'boop axioms' then thought should be given to making it an axiom. In the preceding sections this result was provable for issexp,islist, introduced as an axiom for isint,isset but not even true for isnat.

Then the various functions and predicates which are peculiar to boops are axiomatised paying special care to do so by means of explicit definitions wherever possible.

DISJOINTNESS OF DOMAINS

In the development of the environment so far, nothing has been said about disjointness of lists and integers, say. Before the theories here developed as modules can be used usefully as a unified whole, another axiom must be supplied to insure that any appropriate disjointness is provable.

As an example of what is required in general, we give now an axiom that guarantees the disjointness of integers, S-expressions, sets and beeps:-

$$\begin{aligned}\forall x.\ \mathrm{isint}(x) \rightarrow\ & \mathrm{issexp}(x) \rightarrow UU,\ \mathrm{isset}(x) \rightarrow UU,\ \mathrm{isbeep}(x) \rightarrow UU, x \\ & \mathrm{issexp}(x) \rightarrow\ \mathrm{isset}(x) \rightarrow UU,\ \mathrm{isbeep}(x) \rightarrow UU, x \\ & \mathrm{isset}(x) \rightarrow\ \mathrm{isbeep}(x) \rightarrow UU, x \\ & \mathrm{isbeep}(x) \rightarrow x, UU \qquad \equiv x.\end{aligned}$$

PROJECT STATISTICS.

The total line count for the proofs of the 1000 (approx.) theorems given in the appendices stands at about 20,000 using only those features of 'version 1' LCF (that is the proof checker that is decribed in the 1972 manual [1]. The total cpu time used was about 50 hours and the human effort involved was about 8 man-months (all of which was spent at a time-sharing-system console). The figures for man and computer effort should be interpreted in light of the fact that much of the proving had to be re-done because of a revision of the axioms (After about 15,000 lines of proof some improvements in the axioms were deemed essential and so about 6 man weeks of effort was expended to alter the proofs).

These statistics provide, I believe, a valuable benchmark against which to measure the effectiveness of logics and aids for proof generation. It is proposed in the near future to use at least some of these proofs to gauge some proposed amendments to the input language of the proof checker.

INCOMPLETENESS.

Inspection of the theorems concerning the concept of Integer Primeness immediately reveals that the the ones given are only the trivial properties of 'Pr'. It was also noted in sections 6 and 7 that no properties are given for some of the quite important operations that are defined on lists and sets. There are also, undoubtably, many powerful and useful theorems for the other areas which remain unstated. Although this incompleteness dictates that a user may in certain circumstances be obliged to prove further results, work on expanding the theorem base (for its own sake) has been stopped because the point of diminishing returns has been reached. The future development of this mathematical environment will be accomplished by individuals enunciating theorems as required and supplying the proofs.

Another important reason for only adding (proved) theorems as they are needed is that a new version of the LCF checker will appear (sooner or later) and will incorporate features which will make the task of generating a proof more automatic and so much shorter. There is also the possibility that the typed logic will be replaced by the type free theory proposed by Scott and so the whole treatment would have to be redone (aside: this would take much less than the 8 man- months quoted here because the proof outlines are all done and the proof checker would be better - 3 months is an upper limit).

TO USE THE ENVIRONMENT.

Inevitably some readers will want to make use of theorems from the appendices of this report in the Stanford AI project PDP10 system. The axioms are located in a file called AXIA on [TH,MAL] and the theorems appear in a form which LCF can read in the file THRMS on [TH,MAL]. Note that a large proportion of theorems without assumptions are suitable for immediate inclusion in the SIMPSET (for example ∀X. X+UU ≡ UU) although some (such as the various commutative rules) will cause non-termination of the simplification process. There are actually more theorems in this file than will fit, with LCF, in the 90K of core currently available to jobs in the PDP10 system at Stanford, so the user may have to prune a copy of THRMS to meet his needs. There will shortly be available a core image with a large selection of the most important theorems already read in (and moved to binary program space to reduce garbage collection time).

THEOREM NAMES.

LCF requires a name for every theorem (arbitrary alphanumeric identifier) but provides only one handle for access to a result - its name. Experience immediately suggests to the user that mnemonics will be an important ingredient in the organization of the environment and this is so as examples indicate:-

POS0 - pos(0)≡FF
PLUSUX - ∀X. UU+X≡UU
TIMES0X - isint(X)≡TT ⊢ 0*X≡0
ELTXNS - ∂(X)≡TT ⊢ X∈NS≡FF

However, for the many objects we have, mnemonic tags help only for a small fraction of the cases. Most theorems are not results which have words already associated with them (like associativity) and most have a good number of tokens in the assumptions and conclusion (combined). The author relied on a fairly complex system of mnemonic notions but names tended to be long and absolutely unintelligible to anyone else. What can one do about theorems such as :-

isint(W)≡TT ⊢ (W+X)≥(W+Y) ≡ X≥Y
X⊗Y≡0, isint(W)≡TT ⊢ (X*W)⊗Y≡0
islist(X&Y)≡TT ⊢ islist(Y)≡TT
isset(X)≡TT, ∀W. W∈X≡W∈Y ⊢ X≡Y

to provide mnemonic significance without being so long that typing errors are encouraged unduly? It is apparent that proof generation should be written with more facilities to address theorems by their content and to have appropriate goal-directed procedures to search for the right theorem to apply.

ALGEBRAIC MANIPULATION.

Another situation where proof generation seemed unreasonably tedious was where an expression involving operators which had special properties - commutativity and associativity in particular. A good example of this sort of painful proof ocurred in trying to prove the theorem

$$(X+Y)*(X-Y) \equiv (X*X)-(Y*Y).$$

Ignore the problem of what happens when X or Y are either undefined or simply not integers and suppose isint(X)≡TT, isint(Y)≡TT . The steps in the proof are:-

1)	isint(X*X)≡TT	
2)	(X*X)+0≡X*X	
3)	isint(Y*X)≡TT	
4)	(Y*X)-(Y*X)≡0	
5)	∀X Y Z. (X+Y)-Z≡X+(Y-Z)	
6)	∀X Y Z. (X+Y)*Z≡(X*Z)+(Y*Z)	
7)	∀X Y Z. X-(Y+Z)≡(X-Y)+Z	
8)	∀X Y Z. X+(Y+Z)≡(X+Y)+Z	
9)	((X+Y)*X)-((X+Y)*Y)≡(X*X)-(Y*Y)	(BY 2,4,5:8)
10)	∀X Y Z. X*(Y-Z)≡(X*Y)-(X*Z)	
11)	(X+Y)*(X-Y)≡(X*X)-(Y*Y)	(BY 9,10)

FUTURE WORK

This research has given birth to a lot of suggestions about possible improvements to LCF. Before this mathematical environment is expanded, therefore, a new, more-automatic proof generator should be developed. When a new one is produced, the body of theorems should be reviewed and expanded.

The same sort of experiment is planned to give the same sort of a rigorous theory for a programming language. A suitable language (such as LISP, ALGOL) or a subset of a language will be taken and the semantics axiomatised using LCF. Then important theorems will be formulated and proved as time and imagination permit.

ACKNOWLEDGEMENTS

This work was born out of Richard Weyhrauch's experiments on program correctness and credit is due Robin Milner for getting the LCF project going. I am extremely grateful for the conversations that I had with both of these people throughout the work.

9. REFERENCES

1 - MILNER, R., "Logic for Computable Functions - Description of a Machine Implementation", Artificial Intelligence Memo #169, Computer Science Dept., Stanford University, May 1972.

2 - MILNER, R., "Implementation and Applications of Scott's Logic for Computable Functions", Proc. ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico, Jan 6-7, 1972.

3 - MILNER, R. & WEYHRAUCH, R., "Proving Compiler Correctness in a Mechanised Logic", Machine Intelligence 7, ed. D. Michie, Edinburgh University Press, 1972.

4 - WEYHRAUCH, R., & MILNER, R., "Program Semantics and Correctness in a Mechanised Logic", Proc. USA-Japan Computer Conference Tokyo, Oct 1972.

APPENDIX 1 - Theorems depending on NO axioms.

$\vdash \quad [\lambda X \,.UU] \equiv UU$

$\vdash \quad \forall P \,.\, (P \to TT, FF) \equiv P$

$\vdash \quad \forall P \,.\, (P \to UU, UU) \equiv UU$

$A \subset X,\ B \subset X \ \vdash \quad \forall P \,.\, (P \to A, B) \subset X$

$P \to TT, UU \equiv TT \ \vdash \quad P \equiv TT$

$P \to TT, FF \equiv TT \ \vdash \quad P \equiv TT$

$P \to FF, UU \equiv FF \ \vdash \quad P \equiv TT$

$P \to FF, TT \equiv FF \ \vdash \quad P \equiv TT$

$P \to UU, TT \equiv FF \ \vdash \quad P \equiv FF$

$P \to FF, TT \equiv TT \ \vdash \quad P \equiv FF$

$P \to UU, FF \equiv FF \ \vdash \quad P \equiv FF$

$P \to TT, FF \equiv FF \ \vdash \quad P \equiv FF$

$P \to TT, TT \equiv UU \ \vdash \quad P \equiv UU$

$P \to FF, FF \equiv UU \ \vdash \quad P \equiv UU$

$P \to TT, FF \equiv UU \ \vdash \quad P \equiv UU$

$P \to FF, TT \equiv UU \ \vdash \quad P \equiv UU$

$P \to FF, FF \equiv TT \ \vdash \quad TT \equiv FF$

$P \to FF, UU \equiv TT \ \vdash \quad TT \equiv FF$

$P \to UU, FF \equiv TT \ \vdash \quad TT \equiv FF$

$P \to TT, TT \equiv FF \ \vdash \quad TT \equiv FF$

$P \to TT, UU \equiv FF \ \vdash \quad TT \equiv FF$

$P \to UU, TT \equiv FF \ \vdash \quad TT \equiv FF$

$P(UU) \equiv TT \ \vdash \quad P \equiv [\lambda x \,.TT]$

$P(UU) \equiv FF \ \vdash \quad P \equiv [\lambda x \,.FF]$

APPENDIX 2 - Theorems that follow from the propositional axioms.

$\vdash \neg TT \equiv FF$
$\vdash \neg UU \equiv UU$
$\vdash \neg FF \equiv TT$

$\vdash TT \vee TT \equiv TT$
$\vdash TT \vee UU \equiv TT$
$\vdash TT \vee FF \equiv TT$
$\vdash UU \vee TT \equiv TT$
$\vdash UU \vee UU \equiv UU$
$\vdash UU \vee FF \equiv UU$
$\vdash FF \vee TT \equiv TT$
$\vdash FF \vee UU \equiv UU$
$\vdash FF \vee FF \equiv FF$

$\vdash \forall P.\ TT \vee P \equiv TT$
$\vdash \forall P.\ FF \vee P \equiv P$
$\vdash \forall P.\ P \vee TT \equiv TT$
$\vdash \forall P.\ P \vee FF \equiv P$
$\vdash \forall P.\ UU \vee P \subset TT$
$\vdash \forall P.\ P \vee UU \subset TT$

$\vdash TT \wedge TT \equiv TT$
$\vdash TT \wedge UU \equiv UU$
$\vdash TT \wedge FF \equiv FF$
$\vdash UU \wedge TT \equiv UU$
$\vdash UU \wedge UU \equiv UU$
$\vdash UU \wedge FF \equiv FF$
$\vdash FF \wedge TT \equiv FF$
$\vdash FF \wedge UU \equiv FF$
$\vdash FF \wedge FF \equiv FF$

$\vdash \forall P.\ TT \wedge P \equiv P$
$\vdash \forall P.\ FF \wedge P \equiv FF$
$\vdash \forall P.\ P \wedge TT \equiv P$
$\vdash \forall P.\ P \wedge FF \equiv FF$
$\vdash \forall P.\ UU \wedge P \subset FF$
$\vdash \forall P.\ P \wedge UU \subset FF$

$\vdash TT = TT \equiv TT$
$\vdash TT = UU \equiv UU$
$\vdash TT = FF \equiv FF$
$\vdash UU = TT \equiv UU$
$\vdash UU = UU \equiv UU$
$\vdash UU = FF \equiv UU$
$\vdash FF = TT \equiv FF$
$\vdash FF = UU \equiv UU$
$\vdash FF = FF \equiv TT$

APPENDIX 2 (continued).

$\vdash \forall P.\ UU=P = UU$
$\vdash \forall P.\ P=UU = UU$

$P=Q \equiv TT \vdash P \equiv Q$

$\vdash \forall P.\ \neg(\neg P) \equiv P$
$\vdash P \vee Q \equiv Q \vee P$
$\vdash \forall P\ Q\ R.\ (P \vee Q) \vee R \equiv P \vee (Q \vee R)$
$\vdash P \wedge Q \equiv Q \wedge P$
$\vdash \forall P\ Q\ R.\ (P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$
$\vdash P=Q \equiv Q=P$
$\vdash \forall P\ Q\ R.\ (P=Q)=R \equiv P=(Q=R)$

$P \wedge Q \equiv FF \vdash P \to X, (Q \to Y, Z) \equiv Q \to Y, (P \to X, Z)$
$P \vee Q \equiv FF \vdash P \equiv FF$
$P \vee Q \equiv FF \vdash Q \equiv FF$
$P \wedge Q \equiv TT \vdash P \equiv TT$
$P \wedge Q \equiv TT \vdash Q \equiv TT$

APPENDIX 3 - Theorems that follow from the equality axioms alone.

$\vdash \partial(UU) \equiv UU$

$\vdash \forall X.\ UU=X \equiv UU$

$\vdash \forall X.\ X=UU \equiv UU$

$\partial(X)\equiv UU \vdash X \equiv UU$

$\partial(X)\equiv FF \vdash TT \equiv FF$

$\vdash \forall X\ .\ \partial(X)\rightarrow X, X \equiv X$

$(X=Y)\equiv TT \vdash \partial(X) \equiv TT$

$(X=Y)\equiv FF \vdash \partial(X) \equiv TT$

$\partial(X)\equiv TT \vdash X=X \equiv TT$

$\vdash \forall X\ .\ X=X \equiv \partial(X)$

$(X=Y)\equiv TT \vdash X \equiv Y$

$\partial(X)\equiv TT,\ X \equiv Y \vdash X=Y \equiv TT$

$X=Y\equiv TT,\ Y=Z\equiv TT \vdash X=Z \equiv TT$

$\partial(X)\equiv TT,\ X=Y\equiv UU \vdash Y \equiv UU$

$(X=Y)\equiv TV \vdash Y=X \equiv TV$

$\vdash X=Y \equiv Y=X$

$(X=Y)\equiv FF,\ X<Y \vdash TT \equiv FF$

$\partial(X)\equiv TT,\ X<Y \vdash X \equiv Y$

APPENDIX 4 - Theorems about Natural Numbers (see section 4).

a) Theorems which follow from axioms 4.2 to 4.8 alone:

$\vdash Z(0) \equiv TT$
$\vdash isnat(0) \equiv TT$
$\vdash succ(0) \equiv 1$
$\vdash pred(1) \equiv 0$
$\vdash succ(1) \equiv 2$
$\vdash Z(1) \equiv FF$
$\vdash isnat(1) \equiv TT$
$\vdash pred(2) \equiv 1$
$\vdash Z(2) \equiv FF$
$\vdash isnat(2) \equiv TT$
$\vdash Z(UU) \equiv UU$
$\vdash isnat(UU) \equiv UU$

$Z(X) \equiv TT \vdash X \equiv 0$
$isnat(X) \equiv TT \vdash Z(succ(X)) \equiv FF$
$isnat(X) \equiv TT \vdash isnat(succ(X)) \equiv TT$
$isnat(X) \equiv FF \vdash TT \equiv FF$

$isnat(X) \equiv TT,\ Z(X) \equiv FF \vdash isnat(pred(X)) \equiv TT$
$isnat(X) \equiv TT \vdash pred(succ(X)) \equiv X$
$isnat(X) \equiv TT,\ Z(X) \equiv FF \vdash succ(pred(X)) \equiv X$

$isnat(X) \equiv TT,\ isnat(Y) \equiv TT,\ succ(X) \equiv succ(Y) \vdash X \equiv Y$

$g(0) \equiv TT,\ \forall X.\ isnat(X) :: g(X) :: g(succ(X)) \equiv TT \vdash \forall X.\ isnat(X) :: g(X) \equiv TT$

b) Theorems that use 4.1 to 4.8 and the equality axioms.

$isnat(X) \equiv TT \vdash \partial(X) \equiv TT$
$Z(X) \equiv FF \vdash \partial(X) \equiv TT$
$Z(X) \equiv UU \vdash X \equiv UU$

$\vdash \partial(0) \equiv TT$
$\vdash \partial(1) \equiv TT$
$\vdash \partial(2) \equiv TT$
$\vdash succ(UU) \equiv UU$
$\vdash pred(UU) \equiv UU$
$\vdash (1=0) \equiv FF$
$\vdash (2=0) \equiv FF$
$\vdash (2=1) \equiv FF$

APPENDIX 5 - Proof of an Induction Theorem for Natural Numbers.

```
   [ The proof is as supplied TO the proof checker. ]
   [ material in square brackets is commentary. ]
   [    theorem  TH1 is      Z(x)≡TT ⊢ X≡0
        theorem  TH2 is      ⊢ Z(0)≡TT
        theorem  TH3 is      isnat(x)≡TT,Z(x)≡FF ⊢ isnat(pred(x))≡TT
        theorem  TH4 is      isnat(x)≡TT,Z(x)≡FF ⊢ succ(pred(x))≡x  ]

LABEL L1;
ASSUME g(0)≡TT;
ASSUME ∀X. isnat(X):: g(X):: g(succ(X))≡TT;
GOAL ∀X. isnat(X):: isnat(X):: g(X)≡TT;
TRY INDUCT {step no. of DEF 4.3} OCC 1,3;
TRY 1 SIMPL;
LABEL L2;
 TRY 2 ABSTR;       [ Step .L2 is     ∀X. F(X):: isnat(X):: g(X)≡TT  ]
  TRY 1 CASES Z(X):
   TRY 1 SIMPL;                                          [   Z(X)≡TT   ]
    USE TH1,-;      USE TH2;
    TRY  SIMPL BY -,--,.L1;
   TRY 2 SIMPL;                                          [   Z(X)≡UU   ]
LABEL L3;
   TRY 3 CASES F(pred(X));                               [   Z(X)≡FF   ]
    TRY 2 SIMPL;                                    [   F(pred(X))≡UU   ]
    TRY 3 SIMPL;                                    [   F(pred(X))≡FF   ]
    TRY 1 CASES isnat(X);                           [   F(pred(X))≡TT   ]
     TRY 1 SIMPL;                                     [   isnat(X)≡TT   ]
      USE TH3,-,.L3;                          [   isnat(pred(X))≡TT   ]
      APPL .L2,pred(X);  SIMPL - BY --;             [   g(pred(X))≡TT   ]
      USE TH4,----,.L3;
      APPL .L1+1,pred(X); SIMPL - BY --,---,-----;      [   g(X)≡TT   ]
      TRY SIMPL BY -;
     TRY 2 SIMPL;                                     [   isnat(X)≡UU   ]
     TRY 3 SIMPL;                                     [   isnat(X)≡FF   ]

GOAL ∀X. isnat(X):: g(X)≡TT;
TRY ABSTR;
 TRY 1 CASES isnat(X);
  TRY 1 SIMPL;                                        [   isnat(X)≡TT   ]
   APPL --,X; SIMPL -;
   TRY 1 SIMPL BY -;
  TRY 2 SIMPL;                                        [   isnat(X)≡UU   ]
  TRY 3 SIMPL;                                        [   isnat(X)≡FF   ]
THEOREM MATHIND: -;


          [ The theorem MATHIND is
                g(0)≡TT,  ∀x. isnat(x):: g(x):: g(succ(x))≡TT
                          ⊢ ∀x. isnat(x):: g(x)≡TT      ]
```

APPENDIX 6 - Theorems that follow from axioms 5.1 to 5.8
(together with axioms of sections 3 and 4).

```
   ⊢ pos(0) ≡ FF
   ⊢ pos(1) ≡ TT
   ⊢ pos(2) ≡ TT
   ⊢ pos(UU) ≡ UU

                        ⊢ isint(UU) ≡ UU
isint(X)≡UU             ⊢ X ≡ UU
isint(X)≡TT             ⊢ ∂(X) ≡ TT
pos(X)≡TT               ⊢ isint(X) ≡ TT
pos(X)≡FF               ⊢ isint(X) ≡ TT
isnat(X)≡TT             ⊢ isint(X) ≡ TT
isint(mns(X))≡TT        ⊢ isint(X) ≡ TT
isint(X)≡TT             ⊢ isint(mns(X)) ≡ TT
                        ⊢ isint(0) ≡ TT
                        ⊢ isint(1) ≡ TT
                        ⊢ isint(2) ≡ TT

                  ⊢ mns(0) ≡ 0
isint(X)≡TT       ⊢ mns(mns(X)) ≡ X
                  ⊢ mns(UU) ≡ UU
isint(X)≡FF       ⊢ mns(X) ≡ UU

isint(X)≡FF                       ⊢ Z(X) ≡ FF
pos(X)≡FF, pos(mns(X))≡FF         ⊢ X ≡ 0
pos(X)≡TT                         ⊢ Z(X) ≡ FF
pos(mns(X))≡TT                    ⊢ Z(X) ≡ FF
isnat(X)≡TT, pos(X)≡FF            ⊢ X ≡ 0
                                  ⊢ ∀X. Z(mns(X))≡isint(X)→Z(X),UU

isnat(X)≡TT, Z(X)≡FF      ⊢ pos(X) ≡ TT
isnat(mns(X))≡TT          ⊢ pos(X) ≡ FF
pos(mns(X))≡TT            ⊢ pos(X) ≡ FF
pos(mns(X))≡FF, Z(X)≡FF   ⊢ pos(X) ≡ TT
pos(X)≡TT                 ⊢ pos(mns(X)) ≡ FF
pos(X)≡FF, Z(X)≡FF        ⊢ pos(mns(X)) ≡ TT
isint(X)≡FF               ⊢ pos(X) ≡ UU

Z(mns(X))≡TT  ⊢ X ≡ 0
pos(X)≡TT     ⊢ isnat(X) ≡ TT
pos(X)≡FF     ⊢ isnat(mns(X)) ≡ TT
```

APPENDIX 6 (continued).

```
isint(X)≡FF          ⊢  succ(X) ≡ UU
isint(X)≡FF          ⊢  pred(X) ≡ UU
isint(X)≡TT          ⊢  pred(succ(X)) ≡ X
isint(X)≡TT          ⊢  succ(pred(X)) ≡ X
pos(X)≡TT            ⊢  pos(succ(X)) ≡ TT
pos(X)≡FF            ⊢  pos(pred(X)) ≡ FF
isint(X)≡TT          ⊢  isint(succ(X)) ≡ TT
isint(X)≡TT          ⊢  isint(pred(X)) ≡ TT
isint(succ(X))≡TT    ⊢  isint(X) ≡ TT
isint(pred(X))≡TT    ⊢  isint(X) ≡ TT
                     ⊢  ∀X . succ(mns(X)) ≡ mns(pred(X))
                     ⊢  ∀X . pred(mns(X)) ≡ mns(succ(X))

pos(X)≡UU, isint(X)≡TT   ⊢  TT ≡ FF
mns(X)≡UU, isint(X)≡TT   ⊢  TT ≡ FF
pred(X)≡UU, isint(X)≡TT  ⊢  TT ≡ FF
succ(X)≡UU, isint(X)≡TT  ⊢  TT ≡ FF

g(Ø)≡TT, ∀x.isint(x)::g(x)≡g(succ(x))  ⊢  ∀X. isint(X):: g(X) ≡ TT

g(Ø)≡h(Ø),   ∀X.isint(X)::∂(g(X))≡TT,   ∀X.isint(X)::∂(h(X))≡TT,
  ∀X. isint(X):: (g(X)≡h(X)):: g(succ(X)) ≡ h(succ(X)),
    ∀X. isint(X)::((g(X)≡h(X)):: g(pred(X)) ≡ h(pred(X))   ⊢
                                ∀X. isint(X):: g(X) ≡ h(X)
```

APPENDIX 7 - Theorems about the operations of arithmetic.

(uses the axioms of sections 3, 4 and 5).

a) Consider first the arithmetic of + and -.

```
    ⊢   ∀X. X+UU ≡ UU
    ⊢   ∀X. UU+X ≡ UU
    ⊢   ∀X. X-UU ≡ UU
    ⊢   ∀X. UU-X ≡ UU

isint(X)≡FF  ⊢   ∀Y. X+Y ≡ UU
isint(Y)≡FF  ⊢   ∀X. X+Y ≡ UU
isint(X)≡FF  ⊢   ∀Y. X-Y ≡ UU
isint(Y)≡FF  ⊢   ∀X. X-Y ≡ UU

isint(X)≡TT  ⊢   X+0 ≡ X
isint(X)≡TT  ⊢   X-0 ≡ X
             ⊢   ∀X. X+1 ≡ succ(X)
             ⊢   ∀X. X-1 ≡ pred(X)
isint(X)≡TT  ⊢   X+mns(X) ≡ 0
isint(X)≡TT  ⊢   mns(X)+X ≡ 0
isint(X)≡TT  ⊢   X-X ≡ 0

    ⊢   ∀X Y. succ(X)+pred(Y) ≡ X+Y
    ⊢   ∀X Y. pred(X)+succ(Y) ≡ X+Y
    ⊢   ∀X Y. succ(X)+Y ≡ X+succ(Y)
    ⊢   ∀X Y. pred(X)+Y ≡ X+pred(Y)
    ⊢   ∀X Y. succ(X+Y) ≡ X+succ(Y)
    ⊢   ∀X Y. succ(X+Y) ≡ succ(X)+Y
    ⊢   ∀X Y. pred(X+Y) ≡ X+pred(Y)
    ⊢   ∀X Y. pred(X+Y) ≡ pred(X)+Y

isint(X)≡TT, isint(Y)≡TT  ⊢   isint(X+Y) ≡ TT
isint(X+Y)≡TT             ⊢   isint(X) ≡ TT
isint(X+Y)≡TT             ⊢   isint(Y) ≡ TT

    ⊢   ∀X Y Z. (X+Y)+Z ≡ X+(Y+Z)

isint(X+W)≡TT, X+W≡Y+W  ⊢   X ≡ Y
isint(X)≡TT  ⊢   0+X ≡ X
             ⊢   ∀X. 0-X ≡ mns(X)
             ⊢   ∀X. 1+X ≡ succ(X)
             ⊢   ∀X . 1-X ≡ mns(pred(X))
             ⊢   X+Y ≡ Y+X

    ⊢   ∀X Y. mns(X+Y) ≡ mns(X)+mns(Y)
```

APPENDIX 7 (continued).

```
    ⊢   ∀X Y. succ(X)-Y ≡ X-pred(Y)
    ⊢   ∀X Y. pred(X)-Y ≡ X-succ(Y)
    ⊢   ∀X Y. succ(X)-succ(Y) ≡ X-Y
    ⊢   ∀X Y. pred(X)-pred(Y) ≡ X-Y
    ⊢   ∀X Y. mns(X-Y) ≡ Y-X
    ⊢   ∀X Y Z. X-(Y-Z) ≡ (X-Y)+Z
    ⊢   ∀X Y Z. X-(Y+Z) ≡ (X-Y)-Z
    ⊢   ∀X Y Z. X+(Y-Z) ≡ (X+Y)-Z
    ⊢   ∀X Y. succ(X-Y) ≡ X-pred(Y)
    ⊢   ∀X Y. succ(X-Y) ≡ succ(X)-Y
    ⊢   ∀X Y. pred(X-Y) ≡ X-succ(Y)
    ⊢   ∀X Y . pred(X-Y) ≡ pred(X)-Y

isint(X)≡TT, isint(Y)≡TT  ⊢   isint(X-Y) ≡ TT
isint(X-Y)≡TT             ⊢   isint(X) ≡ TT
isint(X-Y)≡TT             ⊢   isint(Y) ≡ TT

X-Y≡0  ⊢   X ≡ Y
```

b) Now theorems from the defn. of multiplication.

```
                ⊢   ∀X. X*UU ≡ UU
                ⊢   ∀X. UU*X ≡ UU
isint(X)≡FF     ⊢   ∀Y. X*Y ≡ UU
isint(Y)≡FF     ⊢   ∀X. X*Y ≡ UU

isint(X)≡TT     ⊢   X*0 ≡ 0
isint(X)≡TT     ⊢   X*1 ≡ X

isint(X)≡TT, isint(Y)≡TT  ⊢   isint(X*Y) ≡ TT
isint(X*Y)≡TT             ⊢   isint(X) ≡ TT
isint(X*Y)≡TT             ⊢   isint(Y) ≡ TT

    ⊢   ∀X Y. X*Y ≡ (X*pred(Y))+X
    ⊢   ∀X Y. X*succ(Y) ≡ (X*Y)+X
    ⊢   ∀X Y. X*pred(Y) ≡ (X*Y)-X
isint(X)≡TT  ⊢   0*X ≡ 0
    ⊢   ∀X Y. X*Y ≡ (pred(X)*Y)+Y
    ⊢   ∀X Y. succ(X)*Y ≡ (X*Y)+Y
    ⊢   ∀X Y. pred(X)*Y ≡ (X*Y)-Y

    ⊢   X*Y ≡ Y*X
isint(X)≡TT  ⊢   1*X ≡ X
    ⊢   ∀X Y. mns(X)*Y ≡ mns(X*Y)
    ⊢   ∀X Y. X*mns(Y) ≡ mns(X*Y)
    ⊢   ∀X Y. mns(X)*mns(Y) ≡ X*Y
```

APPENDIX 7 (continued).

```
   ⊢   ∀X Y Z. X*(Y+Z) = (X*Y)+(X*Z)
   ⊢   ∀X Y Z. X*(Y-Z) = (X*Y)-(X*Z)
   ⊢   ∀X Y Z. (X+Y)*Z = (X*Z)+(Y*Z)
   ⊢   ∀X Y Z. (X-Y)*Z = (X*Z)-(Y*Z)
   ⊢   ∀X Y Z. (X*Y)*Z = X*(Y*Z)
   ⊢   ∀X Y. (X+Y)*(X-Y) = (X*X)-(Y*Y)

isnat(X)=TT, isnat(Y)=TT        ⊢   isnat(X+Y) = TT
pos(X)=TT, pos(Y)=TT            ⊢   pos(X+Y) = TT
pos(X)=FF, pos(Y)=FF            ⊢   pos(X+Y) = FF
pos(X)=TT, pos(Y)=FF            ⊢   pos(X-Y) = TT
pos(X)=FF, pos(Y)=TT            ⊢   pos(X-Y) = FF
isnat(X)=TT, isnat(Y)=TT        ⊢   isnat(X*Y) = TT
pos(X)=TT, pos(Y)=TT            ⊢   pos(X*Y) = TT
pos(X)=TT, pos(Y)=FF            ⊢   pos(X*Y) = FF
pos(mns(X))=TT, pos(mns(Y))=TT  ⊢   pos(X*Y) = TT
pos(1-X)=TT, isnat(X)=TT        ⊢   X = 0
```

c) Now add the division operator.

```
                ⊢   ∀X. X/UU = UU
                ⊢   ∀X . X/0 = UU
                ⊢   ∀X. UU/X = UU
isint(X)=FF     ⊢   ∀Y. X/Y = UU
isint(Y)=FF     ⊢   ∀X. X/Y = UU

isint(X)=TT, Z(X)=FF      ⊢   0/X = 0
isint(X)=TT, Z(X)=FF      ⊢   X/X = 1
pos(Y-X)=TT, isnat(X)=TT  ⊢   X/Y = 0

∀y. isnat(y):: [αh.[λw.Z(w)→TT,g(pred(w))→h(pred(w)),UU]](y):: g(y)=TT
                                          ⊢   ∀z. isnat(z):: g(z) = TT
  pos(X)=TT, [αh.[λw.Z(w)→TT,f(pred(w))→h(pred(w)),UU]](X)=TT
                        ⊢   ∀Y. isnat(Y):: pos(X-Y):: f(Y) = TT

isnat(X)=TT, pos(Y)=TT              ⊢   isnat(X/Y) = TT
isint(X)=TT, isint(Y)=TT, Z(Y)=FF   ⊢   isint(X/Y) = TT

   ⊢   ∀X Y. mns(X)/Y = mns(X/Y)
   ⊢   ∀X Y. X/mns(Y) = mns(X/Y)
   ⊢   ∀X Y. mns(X)/mns(Y) = X/Y

isint(X/Y)=TT   ⊢   isint(X) = TT
isint(X/Y)=TT   ⊢   Z(Y) = FF
isint(X/Y)=TT   ⊢   isint(Y) = TT

isnat(X)=TT, pos(Y)=TT, isnat(W)=TT  ⊢   ((X*Y)+W)/Y = X+(W/Y)
isint(X)=TT, isint(Y)=TT, Z(Y)=FF    ⊢   (X*Y)/Y = X
```

APPENDIX 7 (continued).
----

d) The mod operator (⊛) is remainder on division.

```
                ⊢  ∀X. X⊛UU ≡ UU
                ⊢  ∀X. X⊛0 ≡ UU
                ⊢  ∀X. UU⊛X ≡ UU
isint(X)≡FF     ⊢  ∀Y. X⊛Y ≡ UU
isint(Y)≡FF     ⊢  ∀X. X⊛Y ≡ UU

isint(X)≡TT, Z(X)≡FF      ⊢  0⊛X ≡ 0
isint(X)≡TT, Z(X)≡FF      ⊢  X⊛X ≡ 0
isnat(X)≡TT, pos(Y-X)≡TT  ⊢  X⊛Y ≡ X

  ⊢  ∀X Y. mns(X)⊛Y ≡ mns(X⊛Y)
  ⊢  ∀X Y. X⊛mns(Y) ≡ X⊛Y
  ⊢  ∀X Y. mns(X)⊛mns(Y) ≡ mns(X⊛Y)

isint(X)≡TT, isint(Y)≡TT, Z(Y)≡FF  ⊢  isint(X⊛Y) ≡ TT
isint(X⊛Y)≡TT                      ⊢  isint(X) ≡ TT
isint(X⊛Y)≡TT                      ⊢  Z(Y) ≡ FF
isint(X⊛Y)≡TT                      ⊢  isint(Y) ≡ TT

isint(X)≡TT, isint(Y)≡TT, Z(Y)≡FF  ⊢  (X⊛Y)⊛Y ≡ 0
isnat(X)≡TT, pos(Y)≡TT, isnat(W)≡TT ⊢  ((X⊛Y)+W)⊛Y ≡ W⊛Y

  ⊢  ∀X Y. X⊛Y ≡ Z(Y)→UU, Z(X)→(isint(Y)→0,UU), (pos(X)→ (pos(Y)→
       (pos(Y-X)→X,(X-Y)⊛Y), X⊛mns(Y)), mns(mns(X)⊛Y) )
  ⊢  ∀X Y. (X⊛Y)⊛Y ≡ X⊛Y
  ⊢  ∀X Y. (X/Y)⊛Y ≡ X-(X⊛Y)

isnat(X)≡TT, isint(Y)≡TT, Z(Y)≡FF ⊢  isnat(X⊛Y) ≡ TT
isint(X)≡TT, isint(Y)≡TT, Z(Y)≡FF ⊢  ((X/Y)⊛Y)+(X⊛Y) ≡ X
isnat(X)≡TT, isnat(Y)≡TT          ⊢  ∀W. (X+Y)⊛W ≡ ((X⊛W)+(Y⊛W))⊛W
(X/W)-(Y/W)≡0, (X⊛W)-(Y⊛W)≡0      ⊢  X ≡ Y

isint(W)≡TT, isint(Y)≡TT, Z(Y)≡FF, W⊛Y≡(W+X)⊛Y  ⊢  X⊛Y ≡ 0
X⊛Y≡0, isint(W)≡TT  ⊢  (X⊛W)⊛Y ≡ 0
X⊛Y≡0, isint(W)≡TT  ⊢  (W⊛X)⊛Y ≡ 0
```

e) Relational operators (> , ≥).

```
  ⊢  ∀X . X≥UU ≡ UU
  ⊢  ∀X . UU≥X ≡ UU
  ⊢  ∀X . X>UU ≡ UU
  ⊢  ∀X . UU>X ≡ UU
```

APPENDIX 7 (continued).

```
isint(X)≡FF ⊢   ∀Y . X≥Y ≡ UU
isint(Y)≡FF ⊢   ∀X . X≥Y ≡ UU
isint(X)≡FF ⊢   ∀Y . X>Y ≡ UU
isint(Y)≡FF ⊢   ∀X . X>Y ≡ UU
X≥Y ≡ TT    ⊢   isint(X) ≡ TT
X≥Y ≡ TT    ⊢   isint(Y) ≡ TT
X>Y ≡ FF    ⊢   isint(X) ≡ TT
X>Y ≡ FF    ⊢   isint(Y) ≡ TT
X>Y ≡ TT    ⊢   X≥Y ≡ TT
X≥Y ≡ FF    ⊢   X>Y ≡ FF
X>X ≡ TT    ⊢   TT ≡ FF
X≥X ≡ FF    ⊢   TT ≡ FF
X>Y≡TT, Y>X≡TT ⊢   TT ≡ FF
X≥Y≡FF, Y≥X≡FF ⊢   TT ≡ FF

isint(X) ≡ TT, isint(Y) ≡ TT, X>Y ≡ UU ⊢   TT ≡ FF
isint(X) ≡ TT, isint(Y) ≡ TT, X≥Y ≡ UU ⊢   TT ≡ FF
X≥Y≡TT, Y≥X≡TT ⊢   X ≡ Y
Y>X ≡ FF       ⊢   X≥Y ≡ TT
Y≥X ≡ FF       ⊢   X>Y ≡ TT
Y>X ≡ TT       ⊢   X≥Y ≡ FF
Y≥X ≡ TT       ⊢   X>Y ≡ FF
W>X≡TT, X>Y≡TT ⊢   W>Y ≡ TT
W≥X≡TT, X>Y≡TT ⊢   W>Y ≡ TT
W>X≡TT, X≥Y≡TT ⊢   W>Y ≡ TT
W≥X≡TT, X≥Y≡TT ⊢   W≥Y ≡ TT
isint(X) ≡ TT  ⊢   X≥X ≡ TT
isint(X) ≡ TT  ⊢   X>X ≡ FF

               ⊢   ∀X . pos(X) ≡ X>0
pos(X) ≡ TT    ⊢   X>0 ≡ TT
X>0 ≡ TT       ⊢   pos(X) ≡ TT
               ⊢   ∀X Y . (X-Y)≥0 ≡ X≥Y
isnat(X-Y) ≡ TT ⊢  X≥Y ≡ TT
isnat(X) ≡ TT  ⊢   X≥0 ≡ TT
isnat(mns(X))≡TT⊢  X>0 ≡ FF
X≥Y ≡ TT       ⊢   isnat(X-Y) ≡ TT
X≥0 ≡ TT       ⊢   isnat(X) ≡ TT
               ⊢   ∀X . pos(X) ≡ 0>mns(X)
0≥X ≡ TT       ⊢   pos(X) ≡ FF
 ⊢   ∀X . X>0 ≡ 0>mns(X)
 ⊢   ∀X . X≥0 ≡ 0≥mns(X)
 ⊢   ∀X Y . mns(X)>mns(Y) ≡ Y>X
 ⊢   ∀X Y . mns(X)≥mns(Y) ≡ Y≥X
 ⊢   ∀X Y . X≥succ(Y) ≡ X>Y
 ⊢   ∀X Y . X>pred(Y) ≡ X≥Y
 ⊢   ∀X Y . pred(X)≥Y ≡ X>Y
 ⊢   ∀X Y . succ(X)>Y ≡ X≥Y
```

APPENDIX 7 (continued).
----

f) The relational operators and arithmetic.

```
isint(X) ≡ TT  ⊢   ∀Y . (X+Y)≥X ≡ Y≥0
isint(Y) ≡ TT  ⊢   ∀X . (X+Y)≥Y ≡ X≥0
isint(X) ≡ TT  ⊢   ∀Y . (X+Y)>X ≡ Y>0
isint(Y) ≡ TT  ⊢   ∀X . (X+Y)>Y ≡ X>0
isint(X) ≡ TT  ⊢   ∀Y . (X-Y)≥X ≡ 0≥Y
isint(X) ≡ TT  ⊢   ∀Y . (X-Y)>X ≡ 0>Y
X>0 ≡ TT       ⊢   ∀Y . (X*Y)≥X ≡ Y≥1
Y>0 ≡ TT       ⊢   ∀X . (X*Y)≥Y ≡ X≥1
X>0 ≡ TT       ⊢   ∀Y . (X*Y)>X ≡ Y>1
Y>0 ≡ TT       ⊢   ∀X . (X*Y)>Y ≡ X>1
X>0≡TT, Y≥1≡TT ⊢   X≥(X/Y) ≡ TT
X>0≡TT, Y>1≡TT ⊢   X>(X/Y) ≡ TT
Y≥0≡TT, X>0≡TT ⊢   X>(Y⊙X) ≡ TT

isint(W) ≡ TT  ⊢   ∀X Y . (X+W)>(Y+W) ≡ X>Y
isint(W) ≡ TT  ⊢   ∀X Y . (W+X)>(W+Y) ≡ X>Y
isint(W) ≡ TT  ⊢   ∀X Y . (X+W)≥(Y+W) ≡ X≥Y
isint(W) ≡ TT  ⊢   ∀X Y . (W+X)≥(W+Y) ≡ X≥Y
isint(W) ≡ TT  ⊢   ∀X Y . (X-W)>(Y-W) ≡ X>Y
isint(W) ≡ TT  ⊢   ∀X Y . (W-X)>(W-Y) ≡ Y>X
isint(W) ≡ TT  ⊢   ∀X Y . (X-W)≥(Y-W) ≡ X≥Y
isint(W) ≡ TT  ⊢   ∀X Y . (W-X)≥(W-Y) ≡ Y≥X
W>0 ≡ TT       ⊢   ∀X Y . (X*W)>(Y*W) ≡ X>Y
W>0 ≡ TT       ⊢   ∀X Y . (W*X)>(W*Y) ≡ X>Y
W>0 ≡ TT       ⊢   ∀X Y . (X*W)≥(Y*W) ≡ X≥Y
W>0 ≡ TT       ⊢   ∀X Y . (W*X)≥(W*Y) ≡ X≥Y
X≥Y≡TT, W>0≡TT ⊢   (X/W)≥(Y/W) ≡ TT
(X/W)>(Y/W) ≡ TT, W>0 ≡ TT     ⊢   X>Y ≡ TT
W>0 ≡ TT, X>0 ≡ TT, Y≥X ≡ TT   ⊢   (W/X)≥(W/Y) ≡ TT
(W/X)>(W/Y)≡TT, W≥0≡TT, Y≥0≡TT ⊢   Y>X ≡ TT

X≥0≡TT, Y≥0≡TT ⊢   (X+Y)≥0 ≡ TT
X>0≡TT, Y>0≡TT ⊢   (X+Y)>0 ≡ TT
X>0≡TT, Y≥0≡TT ⊢   (X+Y)>0 ≡ TT
X≥0≡TT, Y>0≡TT ⊢   (X+Y)>0 ≡ TT
X≥0≡TT, Y≥0≡TT ⊢   (X*Y)≥0 ≡ TT
X>0≡FF, Y>0≡FF ⊢   (X*Y)≥0 ≡ TT
Y>0 ≡ TT       ⊢   ∀X . (X*Y)≥0 ≡ X≥0
Y>0 ≡ TT       ⊢   ∀X . (X*Y)>0 ≡ X>0
0>X≡TT, 0>Y≡TT ⊢   (X*Y)>0 ≡ TT
X≥0≡TT, Y>0≡TT ⊢   (X/Y)≥0 ≡ TT
Y>0 ≡ TT       ⊢   ∀X . (X/Y)>0 ≡ X≥Y
X≥0≡TT, isint(Y)≡TT, Z(Y)≡FF   ⊢   (X⊛Y)≥0 ≡ TT
(X⊛Y)>0 ≡ TT   ⊢   X>0 ≡ TT
```

APPENDIX 7 (continued).

g) The factorial operator.

```
                    ⊢  Fac(UU) ≡ UU
isint(X)≡FF         ⊢  Fac(X) ≡ UU
X≥0 ≡ FF            ⊢  Fac(X) ≡ UU
                    ⊢  Fac(0) ≡ 1
                    ⊢  Fac(1) ≡ 1
                    ⊢  Fac(2) ≡ 2
X≥0 ≡ TT            ⊢  Fac(X)>0 ≡ TT
∂(Fac(X))≡TT        ⊢  X≥0 ≡ TT
X≥0 ≡ TT            ⊢  Fac(X+1) ≡ (X+1)*Fac(X)
X>0 ≡ TT            ⊢  Fac(X)⊗X ≡ 0
Y>0≡TT, X≥Y≡TT      ⊢  Fac(X)⊗Y ≡ 0
Y>0≡TT, X≥Y≡TT      ⊢  Fac(X)⊗Fac(Y) ≡ 0
Y>0≡TT, X>Y≡TT      ⊢  Fac(X)>Fac(Y) ≡ TT
```

h) The oddness and evenness predicates.

```
                    ⊢  even(UU) ≡ UU
                    ⊢  odd(UU) ≡ UU
isint(X) ≡ FF       ⊢  even(X) ≡ UU
isint(X) ≡ FF       ⊢  odd(X) ≡ UU
                    ⊢  even ≡ [λx .(odd(x)→FF,TT)]
                    ⊢  odd ≡ [λx .(even(x)→FF,TT)]
even(X) ≡ TT        ⊢  isint(X) ≡ TT
even(X) ≡ FF        ⊢  isint(X) ≡ TT
odd(X) ≡ TT         ⊢  isint(X) ≡ TT
odd(X) ≡ FF         ⊢  isint(X) ≡ TT
even(X) ≡ UU, isint(X) ≡ TT  ⊢  TT ≡ FF
odd(X) ≡ UU , isint(X) ≡ TT  ⊢  TT ≡ FF
isint(X) ≡ TT       ⊢  even(X*2) ≡ TT
isint(X) ≡ TT       ⊢  even(2*X) ≡ TT
                    ⊢  ∀X . even(mns(X)) ≡ even(X)
                    ⊢  ∀X . odd(mns(X)) ≡ odd(X)
even(X) ≡ TT        ⊢  even(X+1) ≡ FF
```

```
⊢  even(0) ≡ TT
⊢  odd(0) ≡ FF
⊢  even(1) ≡ FF
⊢  odd(1) ≡ TT
⊢  even(2) ≡ TT
⊢  odd(2) ≡ FF
```

APPENDIX 7 (continued).
----

i) The 'Look' operation.

```
P(UU)≡UU  ⊢  ∀F . Look(UU,F,P) ≡ UU
P(X)≡FF   ⊢  Look(X,UU,P) ≡ UU
          ⊢  ∀X F . Look(X,F,UU) ≡ UU
P(X)≡TT   ⊢  ∀F . Look(X,F,P) ≡ X
∀X. P(X)⊂FF      ⊢  ∀X F . Look(X,F,P) ≡ UU
P(X)≡FF, F(X)≡X  ⊢  Look(X,F,P) ≡ UU
```

j) The bounded quantifiers - 'buq' and 'beq'.

```
 ⊢  ∀Y P . buq(UU,Y,P) ≡ UU
 ⊢  ∀X P . buq(X,UU,P) ≡ UU
X>Y ≡ FF        ⊢  buq(X,Y,UU) ≡ UU
isint(X) ≡ FF   ⊢  ∀P . buq(X,Y,P) ≡ UU
isint(Y) ≡ FF   ⊢  ∀P . buq(X,Y,P) ≡ UU
X>Y ≡ TT        ⊢  ∀P . buq(X,Y,P) ≡ TT
isint(X) ≡ TT   ⊢  ∀P . buq(X,X,P) ≡ P(X)
buq(X,Y,P)≡TT   ⊢  isint(X) ≡ TT
buq(X,Y,P)≡TT   ⊢  isint(Y) ≡ TT
buq(X,Y,P)≡FF   ⊢  isint(X) ≡ TT
buq(X,Y,P)≡FF   ⊢  isint(X) ≡ TT

 ⊢  ∀Y P . beq(UU,Y,P) ≡ UU
 ⊢  ∀X P . beq(X,UU,P) ≡ UU
X>Y ≡ FF        ⊢  beq(X,Y,UU) ≡ UU
isint(X) ≡ FF   ⊢  ∀P . beq(X,Y,P) ≡ UU
isint(Y) ≡ FF   ⊢  ∀P . beq(X,Y,P) ≡ UU
X>Y ≡ TT        ⊢  ∀P . beq(X,Y,P) ≡ FF
isint(X) ≡ TT   ⊢  ∀P . beq(X,X,P) ≡ P(X)
beq(X,Y,P)≡TT   ⊢  isint(X) ≡ TT
beq(X,Y,P)≡TT   ⊢  isint(Y) ≡ TT
beq(X,Y,P)≡FF   ⊢  isint(X) ≡ TT
beq(X,Y,P)≡FF   ⊢  isint(X) ≡ TT
```

k) The primeness predicate for integers.

```
                ⊢  Pr(UU) ≡ UU
isint(X) ≡ FF   ⊢  Pr(X) ≡ UU
                ⊢  Pr(0) ≡ FF
                ⊢  Pr(1) ≡ FF
                ⊢  Pr(2) ≡ TT
Pr(X) ≡ TT      ⊢  isint(X) ≡ TT
Pr(X) ≡ FF      ⊢  isint(X) ≡ TT
                ⊢  ∀X . Pr(mns(X)) ≡ Pr(X)
```

APPENDIX 8 - Basic Theorems about S-expressions.

(depends on the equality axioms plus 6.1 - 6.10).

```
 ⊢  issexp(UU) ≡ UU
 ⊢  atom(UU) ≡ UU
 ⊢  null(UU) ≡ UU
 ⊢  head(UU) ≡ UU
 ⊢  tail(UU) ≡ UU

atom(X) ≡ TT   ⊢  head(X) ≡ UU
atom(X) ≡ TT   ⊢  tail(X) ≡ UU
issexp(X) ≡ UU ⊢  X ≡ UU
atom(X) ≡ UU   ⊢  X ≡ UU
null(X) ≡ UU   ⊢  X ≡ UU

 ⊢  issexp(NIL) ≡ TT
 ⊢  ∂(NIL) ≡ TT
 ⊢  null(NIL) ≡ TT
 ⊢  atom(NIL) ≡ TT
 ⊢  head(NIL) ≡ UU
 ⊢  tail(NIL) ≡ UU

issexp(X) ≡ TT ⊢  ∂(X) ≡ TT
issexp(X) ≡ FF ⊢  ∂(X) ≡ TT
atom(X) ≡ TT   ⊢  ∂(X) ≡ TT
atom(X) ≡ FF   ⊢  ∂(X) ≡ TT

null(X) ≡ TT   ⊢  X ≡ NIL
issexp(X) ≡ FF ⊢  null(X) ≡ FF
atom(X) ≡ TT , issexp(X) ≡ TT ⊢  null(X) ≡ TT
atom(X) ≡ FF   ⊢  null(X) ≡ FF

issexp(X) ≡ FF ⊢  atom(X) ≡ TT
issexp(X) ≡ TT , null(X) ≡ FF ⊢  atom(X) ≡ FF
atom(X) ≡ FF   ⊢  issexp(X) ≡ TT
atom(X) ≡ TT , null(X) ≡ FF   ⊢  issexp(X) ≡ FF

∂(head(X)) ≡ TT ⊢  atom(X) ≡ FF
∂(tail(X)) ≡ TT ⊢  atom(X) ≡ FF

 ⊢  ∀X . cons(X,UU) ≡ UU
 ⊢  ∀X . cons(UU,X) ≡ UU

∂(Y) ≡ TT      ⊢  ∀X . head(cons(X,Y)) ≡ X
∂(X) ≡ TT      ⊢  ∀Y . tail(cons(X,Y)) ≡ Y

atom(X) ≡ FF   ⊢  ∂(head(X)) ≡ TT
atom(X) ≡ FF   ⊢  ∂(tail(X)) ≡ TT
```

APPENDIX 8 (continued).

```
head(X) = UU  ⊢   atom(X) ⊂ TT
tail(X) = UU  ⊢   atom(X) ⊂ TT

∂(X) = TT , ∂(Y) = TT  ⊢   issexp(cons(X,Y)) = TT
∂(X) = TT , ∂(Y) = TT  ⊢   null(cons(X,Y)) = FF
∂(X) = TT , ∂(Y) = TT  ⊢   atom(cons(X,Y)) = FF
∂(cons(X,Y)) = TT      ⊢   ∂(X) = TT
∂(cons(X,Y)) = TT      ⊢   ∂(Y) = TT

 ⊢   ∀X , ∂(head(X)) = ∂(tail(X))

head(X) = X  ⊢   X = UU
tail(X) = X  ⊢   X = UU

null(cons(X,Y)) = TT  ⊢   TT = FF
```

APPENDIX 9 - Basic Theorems for Lists.

(axioms used were the equality axioms with C.1 - C.11).

```
⊢  islist(NIL) ≡ TT
⊢  islist(UU) ≡ UU

islist(X) ≡ FF  ⊢  null(X) ≡ FF
issexp(X) ≡ FF  ⊢  islist(X) ≡ FF
islist(X) ≡ TT  ⊢  ∂(X) ≡ TT
islist(X) ≡ FF  ⊢  ∂(X) ≡ TT
islist(X) ≡ TT  ⊢  issexp(X) ≡ TT
islist(X) ≡ TT , null(X) ≡ FF  ⊢  atom(X) ≡ FF
∂(X) ≡ TT       ⊢  ∀Y . islist(cons(X,Y)) ≡ islist(Y)
islist(X) ≡ UU  ⊢  X ≡ UU
islist(tail(X)) ≡ TT  ⊢  islist(X) ≡ TT
islist(X) ≡ TT , null(X) ≡ FF  ⊢  islist(tail(X)) ≡ TT

g(NIL) ≡ TT .
    ∀X Y . ∂(X) :: islist(Y) :: g(Y) :: g(cons(X,Y)) ≡ TT
                          ⊢  ∀X . islist(X) :: g(X) ≡ TT

∀X . atom(X) :: g(X) ≡ TT ,
      ∀X Y . g(X) :: g(Y) :: g(cons(X,Y)) ≡ TT
                    ⊢  ∀X . ∂(X) :: g(X) ≡ TT
```

APPENDIX 10 - Theorems about the list operations of section 6.

(rely on the axioms of section 3 (equality) also).

a) Concerning `rev' and the auxiliary function `rev2'.

```
 ⊢  ∀X . rev2(UU,X) ≡ UU
 ⊢  rev(UU) ≡ UU
 ⊢  ∀X . rev2(X,UU) ≡ UU
 ⊢  ∀X . rev2(NIL,X) ≡ X
 ⊢  ∀X . rev2(X,NIL) ≡ rev(X)
 ⊢  rev(NIL) ≡ NIL
islist(X) ≡ FF   ⊢  ∀Y . rev2(X,Y) ≡ UU
islist(X) ≡ FF   ⊢  rev(X) ≡ UU
islist(X) ≡ TT , ∂(Y) ≡ TT  ⊢  ∂(rev2(X,Y)) ≡ TT
islist(X) ≡ TT   ⊢  ∂(rev(X)) ≡ TT
∂(rev2(X,Y)) ≡ TT ⊢  islist(X) ≡ TT
∂(rev2(X,Y)) ≡ TT ⊢  ∂(Y) ≡ TT
∂(rev(X)) ≡ TT   ⊢  islist(X) ≡ TT
islist(X) ≡ TT , islist(Y) ≡ TT  ⊢  rev(rev2(X,Y)) ≡ rev2(Y,X)
islist(X) ≡ TT   ⊢  rev(rev(X)) ≡ X
islist(X) ≡ TT   ⊢  ∀Y . islist(rev2(X,Y)) ≡ islist(Y)
islist(X) ≡ TT   ⊢  islist(rev(X)) ≡ TT
 ⊢  ∀X . rev(cons(X,NIL)) ≡ cons(X,NIL)
 ⊢  ∀X Y . rev(cons(X,cons(Y,NIL))) ≡ cons(Y,cons(X,NIL))
islist(X) ≡ TT   ⊢  null(rev(X)) ≡ null(X)
```

b) Concerning the `&' (append) function.

```
                ⊢  ∀X . UU&X ≡ UU
                ⊢  ∀X . X&UU ≡ UU
islist(X)≡FF    ⊢  ∀Y . X&Y ≡ UU
                ⊢  ∀X . NIL&X ≡ X
islist(X)≡TT    ⊢  X&NIL ≡ X
                ⊢  ∀X Y . X&Y ≡ rev2(rev(X),Y)
islist(X)≡TT, ∂(Y)≡TT  ⊢  ∂(X&Y) ≡ TT
islist(X)≡TT    ⊢  ∀Y . islist(X&Y) ≡ islist(Y)
                ⊢  ∀X Y . cons(X,NIL)&Y ≡ cons(X,Y)
                ⊢  ∀X Y . rev(X&Y) ≡ rev(Y)&rev(X)
                ⊢  ∀X Y . rev(X&cons(Y,NIL)) ≡ cons(Y,rev(X))
islist(X)≡TT, ∂(Y)≡TT  ⊢  head(X&Y) ≡ null(X)→head(Y),head(X)
islist(X)≡TT    ⊢  tail(X&Y) ≡ null(X)→tail(Y),(tail(X)&Y)
∂(X&Y) ≡ TT     ⊢  islist(X) ≡ TT
∂(X&Y) ≡ TT     ⊢  ∂(Y) ≡ TT
islist(X)≡TT, null(X)≡FF, ∂(Y)≡TT  ⊢  null(X&Y) ≡ FF
islist(X)≡TT, null(Y)≡FF           ⊢  null(X&Y) ≡ FF
X&Y ≡ NIL       ⊢  X ≡ NIL
X&Y ≡ NIL       ⊢  Y ≡ NIL
                ⊢  ∀X Y Z . (X&Y)&Z ≡ X&(Y&Z)
```

APPENDIX 10 (continued).
----

c) Properties of 'ANDmap' and 'ORmap',

```
                ⊢  ∀p . ANDmap(UU,p) = UU
islist(X)=FF    ⊢  ∀p . ANDmap(X,p) = UU
                ⊢  ∀p . ORmap(UU,p) = UU
islist(X)=FF    ⊢  ∀p . ORmap(X,p) = UU
p(X) = UU       ⊢  ∀Y . ANDmap(cons(X,Y),p) = UU
p(X) = UU       ⊢  ∀Y . ORmap(cons(X,Y),p) = UU

                ⊢  ∀p . ANDmap(NIL,p) = TT
                ⊢  ∀p . ORmap(NIL,p) = FF
∂(X) = TT       ⊢  ∀p . ANDmap(cons(X,NIL),p) = p(X)
∂(X) = TT       ⊢  ∀p . ORmap(cons(X,NIL),p) = p(X)

ANDmap(X,p)→TT,TT = TT  ⊢  islist(X) = TT
ORmap(X,p)→TT,TT = TT   ⊢  islist(X) = TT
ANDmap(X,p) = FF        ⊢  null(X) = FF
ORmap(X,p) = TT         ⊢  null(X) = FF
ANDmap(X,p) = TT, p(X) = TT, ∂(X) = TT  ⊢  ANDmap(cons(X,Y),p)=TT
p(X) = FF, islist(cons(X,Y)) = TT       ⊢  ANDmap(cons(X,Y),p)=FF
ANDmap(Y,p) = FF, p(X)→∂(X),∂(X) = TT   ⊢  ANDmap(cons(X,Y),p)=FF
ORmap(Y,p) = FF, p(X) = FF, ∂(X) = TT   ⊢  ORmap(cons(X,Y),p)=FF
p(X) = TT, islist(cons(X,Y)) = TT       ⊢  ORmap(cons(X,Y),p)=TT
ORmap(Y,p) = TT, p(X)→∂(X),∂(X) = TT    ⊢  ORmap(cons(X,Y),p)=TT

∀X.∂(X)::p(X)→TT,TT=TT, islist(Y)=TT    ⊢  ANDmap(Y,p)→TT,TT=TT
∀X.∂(X)::p(X)→TT,TT=TT, islist(Y)=TT    ⊢  ORmap(Y,p)→TT,TT=TT
ANDmap(Y,p)→TT,TT = TT, p(X)→∂(X),∂(X) = TT
                                        ⊢  ANDmap(cons(X,Y),p)→TT,TT = TT
ORmap(Y,p)→TT,TT = TT, p(X)→∂(X),∂(X) = TT
                                        ⊢  ORmap(cons(X,Y),p)→TT,TT = TT
ANDmap(X,p) = TT, null(X)=FF            ⊢  p(head(X)) = TT
ANDmap(X,p) = TT, null(X)=FF            ⊢  ANDmap(tail(X),p) = TT
ORmap(X,p) = FF, null(X)=FF             ⊢  p(head(X)) = FF
ORmap(X,p) = FF, null(X)=FF             ⊢  ORmap(tail(X),p) = FF
ANDmap(X,p)→TT,TT=TT,null(X)=FF         ⊢  p(head(X))→TT,TT = TT
ORmap(X,p)→TT,TT=TT, null(X)=FF         ⊢  p(head(X))→TT,TT = TT

ANDmap(X,p)=TT, ANDmap(Y,p)=TT          ⊢  ANDmap(rev2(X,Y),p) = TT
ORmap(X,p)=FF, ORmap(Y,p)=FF            ⊢  ORmap(rev2(X,Y),p) = FF
ANDmap(X,p)=TT, ANDmap(Y,p)=TT          ⊢  ANDmap(X&Y,p) = TT
ORmap(X,p)=FF, ORmap(Y,p)=FF            ⊢  ORmap(X&Y,p) = FF
ANDmap(X,p) = TT  ⊢  ANDmap(rev(X),p) = TT
ORmap(X,p) = FF   ⊢  ORmap(rev(X),p) = FF
ANDmap(X&Y,p)=TT  ⊢  ANDmap(X,p) = TT
ANDmap(X&Y,p)=TT  ⊢  ANDmap(Y,p) = TT
ORmap(X&Y,p)=FF   ⊢  ORmap(X,p) = FF
ORmap(X&Y,p)=FF   ⊢  ORmap(Y,p) = FF
```

APPENDIX 10 (continued).

```
ANDmap(rev(X),p) ≡ TT  ⊢  ANDmap(X,p) ≡ TT
ORmap(rev(X),p) ≡ FF   ⊢  ORmap(X,p) ≡ FF

ANDmap(X,p)≡FF, islist(Y)≡TT, ∀X. ∂(X):: p(X)→TT,TT ≡ TT
                       ⊢  ANDmap(X&Y,p) ≡ FF
ANDmap(Y,p)≡FF, islist(X)≡TT, ∀X. ∂(X):: p(X)→TT,TT ≡ TT
                       ⊢  ANDmap(X&Y,p) ≡ FF
ORmap(X,p)≡TT, islist(Y)≡TT, ∀X. ∂(X):: p(X)→TT,TT ≡ TT
                       ⊢  ORmap(X&Y,p) ≡ TT
ORmap(Y,p)≡TT, islist(X)≡TT, ∀X. ∂(X):: p(X)→TT,TT ≡ TT
                       ⊢  ORmap(X&Y,p) ≡ TT
ANDmap(X,p)≡FF, ∀X. ∂(X):: p(X)→TT,TT ≡ TT
                       ⊢  ANDmap(rev(X),p) ≡ FF
ORmap(X,p)≡TT, ∀X. ∂(X):: p(X)→TT,TT ≡ TT
                       ⊢  ORmap(rev(X),p) ≡ TT
```

d) Theorems concerning the 'FNmap' function.

```
                ⊢  ∀f . FNmap(UU,f) ≡ UU
islist(X)≡FF    ⊢  ∀f . FNmap(X,f) ≡ UU
                ⊢  ∀f . FNmap(NIL,f) ≡ NIL
∂(X) ≡ TT       ⊢  FNmap(cons(X,NIL),f) ≡ cons(f(X),NIL)
∂(FNmap(X,f))≡TT  ⊢  islist(X) ≡ TT
null(FNmap(X,f))≡FF ⊢  null(X) ≡ FF
null(FNmap(X,f))≡TT ⊢  null(X) ≡ TT
∀X. ∂(X):: ∂(f(X))≡TT, islist(X)≡TT  ⊢  ∂(FNmap(X,f)) ≡ TT

 ⊢  ∀X f. islist(FNmap(X,f))≡∂(FNmap(X,f))
 ⊢  ∀X Y f . FNmap(X&Y,f) ≡ FNmap(X,f)&FNmap(Y,f)
 ⊢  ∀X f . FNmap(rev(X),f) ≡ rev(FNmap(X,f))
```

e) Properties of the 'PRUNE' function.

```
                ⊢  ∀p . PRUNE(UU,p) ≡ UU
islist(X)≡FF    ⊢  ∀p . PRUNE(X,p) ≡ UU
                ⊢  ∀p . PRUNE(NIL,p) ≡ NIL
p(X)≡TT, ∂(X)≡TT  ⊢  PRUNE(cons(X,NIL),p) ≡ NIL
p(X)≡FF, ∂(X)≡TT  ⊢  PRUNE(cons(X,NIL),p) ≡ cons(X,NIL)
∂(PRUNE(X,p))≡TT  ⊢  islist(X) ≡ TT
null(PRUNE(X,p)) ≡ FF  ⊢  null(X) ≡ FF
∀X. ∂(X):: p(X)→TT,TT ≡ TT, islist(X)≡TT  ⊢  ∂(PRUNE(X,p))≡TT

 ⊢  ∀X p . islist(PRUNE(X,p)) ≡ ∂(PRUNE(X,p))
 ⊢  ∀X Y p . PRUNE(X&Y,p) ≡ PRUNE(X,p)&PRUNE(Y,p)
 ⊢  ∀X p . PRUNE(rev(X),p) ≡ rev(PRUNE(X,p))
```

APPENDIX 10 (continued).
-------- --

f) The 'mem' predicate.

```
 ⊢   ∀X . mem(UU,X) ≡ UU
 ⊢   ∀X . mem(X,UU) ≡ UU
islist(Y)≡FF  ⊢   ∀X . mem(X,Y) ≡ UU
islist(Y)≡TT, mem(X,Y)≡UU  ⊢   X ≡ UU
mem(X,Y)≡TT   ⊢   ∂(X) ≡ TT
mem(X,Y)≡FF   ⊢   ∂(X) ≡ TT
mem(X,Y)≡TT   ⊢   islist(Y) ≡ TT
mem(X,Y)≡FF   ⊢   islist(Y) ≡ TT
mem(X,Y)≡TT   ⊢   null(Y) ≡ FF
∂(X) ≡ TT     ⊢   mem(X,NIL) ≡ FF

∂(X)≡TT, islist(Y)≡TT  ⊢   mem(X,cons(X,Y)) ≡ TT
mem(X,cons(Y,NIL))≡TT  ⊢   X ≡ Y
(X=head(Y)) ≡ FF       ⊢   mem(X,tail(Y)) ≡ mem(X,Y)
∀X. ∂(X):: mem(X,Y)≡FF ⊢   Y ≡ NIL
mem(X,Y)≡TT, ∂(W)≡TT   ⊢   mem(X,cons(W,Y)) ≡ TT
islist(tail(X)) ≡ TT   ⊢   mem(head(X),X) ≡ TT
mem(X,Y)≡FF,null(Y)≡FF ⊢   (X=head(Y)) ≡ FF
mem(X,Y)≡FF,null(Y)≡FF ⊢   mem(X,tail(Y)) ≡ FF
                       ⊢   ∀X Y . mem(X,rev(Y)) ≡ mem(X,Y)
mem(X,Y1)≡TT, islist(Y2)≡TT  ⊢   mem(X,(Y1&Y2)) ≡ TT
mem(X,Y2)≡TT, islist(Y1)≡TT  ⊢   mem(X,(Y1&Y2)) ≡ TT
mem(X,(Y1&Y2)) ≡ FF          ⊢   mem(X,Y1) ≡ FF
mem(X,(Y1&Y2)) ≡ FF          ⊢   mem(X,Y2) ≡ FF
mem(X,Y1)≡FF, mem(X,Y2)≡FF   ⊢   mem(X,(Y1&Y2)) ≡ FF

 ⊢   mem ≡ [αG.[λx y .(islist(y)→
       (null(y)→(∂(x)→FF,UU),((x=head(y))→TT,G(x,tail(y)))),UU)]]
```

g) The 'memL' predicate.

```
 ⊢   ∀X . memL(UU,X) ≡ UU
 ⊢   ∀X . memL(X,UU) ≡ UU
islist(X) ≡ FF  ⊢   ∀Y . memL(X,Y) ≡ UU
islist(Y) ≡ FF  ⊢   ∀X . memL(X,Y) ≡ UU
memL(X,Y) ≡ TT  ⊢   islist(X) ≡ TT
memL(X,Y) ≡ FF  ⊢   islist(X) ≡ TT
memL(X,Y) ≡ TT  ⊢   islist(Y) ≡ TT
memL(X,Y) ≡ FF  ⊢   islist(Y) ≡ TT
islist(X) ≡ TT  ⊢   memL(NIL,X) ≡ TT
islist(X) ≡ TT , islist(Y) ≡ TT , memL(X,Y) ≡ UU  ⊢   TT ≡ FF

 ⊢   ∀X Y . memL(cons(X,NIL),Y) ≡ mem(X,Y)
```

APPENDIX 10 (continued).
-------- --

```
memL(tail(X),Y) ≡ TT      ⊢  memL(X,Y) ≡ mem(head(X),Y)
memL(X,Y)≡TT, null(X)≡FF  ⊢  mem(head(X),Y) ≡ TT
memL(X,Y)≡TT, null(X)≡FF  ⊢  memL(tail(X),Y) ≡ TT
memL(tail(X),Y) ≡ FF      ⊢  memL(X,Y) ≡ FF
memL(X,Y)≡TT, mem(A,X)≡TT ⊢  mem(A,Y) ≡ TT

 ⊢  memL ≡ [αG.[λx y .(islist(y)→(islist(x)→
      (null(x)→TT,(mem(head(x),y)→G(tail(x),y),FF)),UU),UU)]]
memL(X,tail(Y)) ≡ TT      ⊢  memL(X,Y) ≡ TT
null(Y)≡FF, memL(X,Y)≡FF  ⊢  memL(X,tail(Y)) ≡ FF
islist(X) ≡ TT            ⊢  memL(X,X) ≡ TT
islist(X)≡TT, islist(Y)≡TT, ∀A. mem(A,X):: mem(A,Y) ≡ TT
                          ⊢  memL(X,Y) ≡ TT
∀X. islist(X):: memL(X,Y)≡null(X)  ⊢  Y ≡ NIL
memL(X,NIL) ≡ TT          ⊢  null(X) ≡ TT
memL(W,X)≡TT,memL(X,Y)≡TT ⊢  memL(W,Y) ≡ TT
                          ⊢  ∀X Y . memL(rev(X),Y) ≡ memL(X,Y)
                          ⊢  ∀X Y . memL(X,rev(Y)) ≡ memL(X,Y)
memL(X,L1)≡TT, islist(L2)≡TT ⊢  memL(X,L1&L2) ≡ TT
memL(X,L2)≡TT, islist(L1)≡TT ⊢  memL(X,L1&L2) ≡ TT
memL(X1,Y)≡TT, memL(X2,Y)≡TT ⊢  memL(X1&X2,Y) ≡ TT
memL(X1&X2,Y) ≡ TT        ⊢  memL(X1,Y) ≡ TT
memL(X1&X2,Y) ≡ TT        ⊢  memL(X2,Y) ≡ TT
memL(X,Y1&Y2) ≡ FF        ⊢  memL(X,Y1) ≡ FF
memL(X,Y1&Y2) ≡ FF        ⊢  memL(X,Y2) ≡ FF
memL(X1,Y)≡FF, islist(X2)≡TT ⊢  memL(X1&X2,Y) ≡ FF
memL(X2,Y)≡F , islist(X1)≡TT ⊢  memL(X1&X2,Y) ≡ FF
```

h) `memEQ' - Equality with respect to (list) membership.

```
 ⊢  ∀X . memEQ(UU,X) ≡ UU
 ⊢  ∀X . memEQ(X,UU) ≡ UU
islist(X) ≡ FF  ⊢  ∀Y . memEQ(X,Y) ≡ UU
islist(Y) ≡ FF  ⊢  ∀X . memEQ(X,Y) ≡ UU
memEQ(X,Y)≡ TT  ⊢  islist(X) ≡ TT
memEQ(X,Y)≡ FF  ⊢  islist(X) ≡ TT
memEQ(X,Y)≡ TT  ⊢  islist(Y) ≡ TT
memEQ(X,Y)≡ FF  ⊢  islist(Y) ≡ TT
islist(X)≡TT, islist(Y)≡TT, memEQ(X,Y)≡UU  ⊢  TT ≡ FF
memEQ(X,Y)≡ TT  ⊢  memL(X,Y) ≡ TT
memEQ(X,Y)≡ TT  ⊢  memL(Y,X) ≡ TT
memL(X,Y) ≡ FF  ⊢  memEQ(X,Y) ≡ FF
memL(Y,X) ≡ FF  ⊢  memEQ(X,Y) ≡ FF
islist(X) ≡ TT  ⊢  memEQ(X,X) ≡ TT
islist(X) ≡ TT  ⊢  memEQ(X,rev(X)) ≡ TT
```

APPENDIX 10 (continued).
-------- --

```
   ⊢   ∀X Y . memEQ(X,Y) ≡ memEQ(Y,X)
memEQ(W,X)≡TT, memEQ(X,Y)≡TT ⊢   memEQ(W,Y) ≡ TT
memEQ(W,X)≡TT, memEQ(X,Y)≡FF ⊢   memEQ(W,Y) ≡ FF
memEQ(X,Y) ≡ TT               ⊢   memEQ(X&Y,X) ≡ TT
memEQ(X,Y) ≡ TT               ⊢   memEQ(X&Y,Y) ≡ TT
memEQ(X,Y) ≡ TT               ⊢   ∀z. mem(z,X) ≡ mem(z,Y)
islist(X)≡TT, ∀z. mem(z,X)≡mem(z,Y)  ⊢   memEQ(X,Y) ≡ TT
```

i) The `memS' operation (deleting an element from a list).

```
                  ⊢   ∀X . memS(UU,X) ≡ UU
                  ⊢   ∀X . memS(X,UU) ≡ UU
islist(X) ≡ FF    ⊢   ∀Y . memS(X,Y) ≡ UU
∂(memS(X,Y))≡TT   ⊢   islist(X) ≡ TT
∂(memS(X,Y))≡TT   ⊢   ∂(Y) ≡ TT
islist(X)≡TT, ∂(Y)≡TT  ⊢   islist(memS(X,Y)) ≡ TT
∂(X) ≡ TT         ⊢   memS(NIL,X) ≡ NIL
                  ⊢   ∀X Y . memS(cons(Y,X),Y) ≡ memS(X,Y)
islist(X)≡TT, ∂(Y)≡TT  ⊢   mem(Y,memS(X,Y)) ≡ FF
islist(X)≡TT, ∂(Y)≡TT  ⊢   memL(memS(X,Y),X) ≡ TT
mem(Y,X) ≡ FF     ⊢   memS(X,Y) ≡ X
                  ⊢   ∀X Y . (memS(X,Y)≡X) ≡ (mem(Y,X)→FF,TT)
                  ⊢   ∀X Y . memL(X,memS(X,Y)) ≡ (mem(Y,X)→FF,TT)
                  ⊢   ∀X Y . memEQ(memS(X,Y),X) ≡ (mem(Y,X)→FF,TT)
```

i) The `memSL' operation.

```
                  ⊢   ∀X . memSL(UU,X) ≡ UU
                  ⊢   ∀X . memSL(X,UU) ≡ UU
islist(X) ≡ FF    ⊢   ∀Y . memSL(X,Y) ≡ UU
islist(Y) ≡ FF    ⊢   ∀X . memSL(X,Y) ≡ UU
∂(memSL(X,Y))≡TT  ⊢   islist(X) ≡ TT
∂(memSL(X,Y))≡TT  ⊢   islist(Y) ≡ TT
islist(X)≡TT, islist(Y)≡TT  ⊢   islist(memSL(X,Y)) ≡ TT
islist(X) ≡ TT    ⊢   memSL(NIL,X) ≡ NIL
islist(X) ≡ TT    ⊢   memSL(X,NIL) ≡ X

islist(X) ≡ TT    ⊢   ∀W Y. mem(W,memSL(X,Y))≡(mem(W,Y)→FF,mem(W,X))
mem(W,Y)≡TT, islist(X)≡TT   ⊢   mem(W,memSL(X,Y)) ≡ FF
mem(W,X)≡FF, islist(Y)≡TT   ⊢   mem(W,memSL(X,Y)) ≡ FF
mem(W,X)≡TT, mem(W,Y)≡FF    ⊢   mem(W,memSL(X,Y)) ≡ TT
mem(W,memSL(X,Y)) ≡ TT      ⊢   mem(W,X) ≡ TT
mem(W,memSL(X,Y)) ≡ TT      ⊢   mem(W,Y) ≡ FF
islist(X) ≡ TT              ⊢   memSL(X,X) ≡ NIL
```

APPENDIX 10 (continued).
-------- --

k) Properties of `subexp'.

```
                          ⊢  ∀X . subexp(X,UU) ≡ UU
                          ⊢  ∀X . subexp(UU,X) ≡ UU
   subexp(X,Y) ≡ TT       ⊢  ∂(X) ≡ TT
   subexp(X,Y) ≡ TT       ⊢  ∂(Y) ≡ TT
   subexp(X,Y) ≡ FF       ⊢  ∂(X) ≡ TT
   subexp(X,Y) ≡ FF       ⊢  ∂(Y) ≡ TT
   ∂(X)≡TT, ∂(Y)≡TT, subexp(X,Y)≡UU  ⊢  TT ≡ FF
   ∂(X) ≡ TT              ⊢  subexp(X,X) ≡ TT
   atom(X) ≡ FF           ⊢  subexp(head(X),X) ≡ TT
   atom(X) ≡ FF           ⊢  subexp(tail(X),X) ≡ TT
   atom(Y) ≡ TT           ⊢  ∀X . subexp(X,Y) ≡ (X≡Y)
   ∂(X) ≡ TT              ⊢  ∀Y . subexp(X,cons(X,Y)) ≡ ∂(Y)
   ∂(X) ≡ TT              ⊢  ∀Y . subexp(Y,cons(X,Y)) ≡ ∂(Y)
   subexp(X,head(Y))≡TT   ⊢  subexp(X,Y) ≡ TT
   subexp(X,tail(Y))≡TT   ⊢  subexp(X,Y) ≡ TT
   subexp(W,X)≡TT, subexp(X,Y)≡TT    ⊢  subexp(W,Y) ≡ TT
   subexp(head(X),Y)≡FF   ⊢  subexp(X,Y) ≡ FF
   subexp(tail(X),Y)≡FF   ⊢  subexp(X,Y) ≡ FF
   subexp(X,Y)≡FF, atom(Y)≡FF        ⊢  subexp(X,head(Y)) ≡ FF
   subexp(X,Y)≡FF, atom(Y)≡FF        ⊢  subexp(X,tail(Y)) ≡ FF
   subexp(X,Y)≡TT, subexp(Y,X)≡TT    ⊢  X ≡ Y
   atom(X) ≡ FF           ⊢  subexp(X,head(X)) ≡ FF
   atom(X) ≡ FF           ⊢  subexp(X,tail(X)) ≡ FF
```

l) Properties of `assoc'.

```
                    ⊢  ∀X . assoc(X,UU) ≡ UU
                    ⊢  ∀X . assoc(UU,X) ≡ UU
   islist(Y) ≡ FF   ⊢  ∀X . assoc(X,Y) ≡ UU
   atom(X) ≡ TT     ⊢  ∀W Y . assoc(W,cons(X,Y)) ≡ UU
   ∂(assoc(X,Y))≡TT ⊢  ∂(X) ≡ TT
   ∂(assoc(X,Y))≡TT ⊢  ∂(Y) ≡ TT
   ∂(X) ≡ TT        ⊢  assoc(X,NIL) ≡ NIL
   islist(Y) ≡ TT   ⊢  ∀W X.assoc(W,cons(cons(W,X),Y))≡cons(W,X)
```

m) The `forL' function.

```
                      ⊢  ∀f fNIL . forL(UU,f,fNIL) ≡ UU
   ∀X. f(X,UU)≡UU, islist(X)≡FF  ⊢  ∀fNIL . forL(X,f,fNIL) ≡ UU
   ∂(forL(X,f,fNIL))≡TT⊢  ∂(X) ≡ TT
                      ⊢  ∀f fNIL. forL(NIL,f,fNIL) ≡ fNIL
   ∂(X) ≡ TT          ⊢  ∀f fNIL. forL(cons(X,NIL),f,fNIL)≡f(X,fNIL)
   ∂(X)≡TT, ∂(Y)≡TT
        ⊢  ∀f fNIL. forL(cons(X,cons(Y,NIL)),f,fNIL)≡f(X,f(Y,fNIL))
```

APPENDIX 11 - Basic Theorems for Finite Sets

(uses the axioms of sections 3,6 and 7.1 to 7.5)

```
                    ⊢  isset(UU) ≡ UU
isset(X)≡UU         ⊢  X ≡ UU
isset(X)≡TT         ⊢  ∂(X) ≡ TT
isset(X)≡FF         ⊢  ∂(X) ≡ TT

                    ⊢  setof(UU) ≡ UU
                    ⊢  listof(UU) ≡ UU
islist(X)≡FF        ⊢  setof(X) ≡ UU
isset(X)≡FF         ⊢  listof(X) ≡ UU
islist(X)≡TT        ⊢  isset(setof(X)) ≡ TT
isset(X)≡TT         ⊢  islist(listof(X)) ≡ TT
isset(X)≡TT         ⊢  setof(listof(X)) ≡ X
∂(setof(X))≡TT      ⊢  islist(X) ≡ TT
∂(listof(X))≡TT     ⊢  isset(X) ≡ TT

memEQ(X,Y)≡TT       ⊢  setof(X) ≡ setof(Y)
                    ⊢  ∀X . setof(listof(setof(X))) ≡ setof(X)
                    ⊢  ∀X . listof(setof(listof(X))) ≡ listof(X)
islist(X)≡TT        ⊢  memEQ(X,listof(setof(X))) ≡ TT
                    ⊢  ∀X L . mem(X,listof(setof(L))) ≡ mem(X,L)
```

APPENDIX 12 - Theorems About the Basic Set Operations.

(relies on the axioms of sections 3,6,7).

a) Theorems involving the null set.

```
 ⊢   isset(NS) ≡ TT
 ⊢   ∂(NS) ≡ TT
 ⊢   listof(NS) ≡ NIL
setof(X) ≡ NS          ⊢   X ≡ NIL
listof(X) ≡ NIL        ⊢   X ≡ NS
isset(X)≡TT, (X≠NS)≡FF ⊢   null(listof(X)) ≡ FF
```

b) Properties of the membership relation.

```
                       ⊢   ∀X . X∈UU ≡ UU
                       ⊢   ∀X . UU∈X ≡ UU
isset(Y)≡FF            ⊢   ∀X . X∈Y ≡ UU
isset(Y)≡TT,X∈Y≡UU     ⊢   X ≡ UU
X∈Y≡TT                 ⊢   ∂(X) ≡ TT
X∈Y≡FF                 ⊢   ∂(X) ≡ TT
X∈Y≡TT                 ⊢   isset(Y) ≡ TT
X∈Y≡FF                 ⊢   isset(Y) ≡ TT
∂(X) ≡ TT              ⊢   X∈NS ≡ FF
∀X. ∂(X):: X∈Y≡FF      ⊢   Y ≡ NS
isset(Y)≡TT, ∀X. X∈Y2≡X∈Y  ⊢   Y2 ≡ Y
```

c) Introducing the 'subset' relation.

```
                   ⊢   ∀X . subset(X,UU) ≡ UU
                   ⊢   ∀X . subset(UU,X) ≡ UU
isset(X) ≡ FF      ⊢   ∀Y . subset(X,Y) ≡ UU
isset(Y) ≡ FF      ⊢   ∀X . subset(X,Y) ≡ UU
subset(X,Y)≡TT     ⊢   isset(X) ≡ TT
subset(X,Y)≡TT     ⊢   isset(Y) ≡ TT
subset(X,Y)≡FF     ⊢   isset(X) ≡ TT
subset(X,Y)≡FF     ⊢   isset(Y) ≡ TT
isset(X)≡TT, isset(Y)≡TT, subset(X,Y)≡UU  ⊢   TT ≡ FF
isset(X)≡TT        ⊢   subset(NS,X) ≡ TT
subset(X,NS)≡TT    ⊢   X ≡ NS
subset(X,Y)≡TT, W∈X≡TT  ⊢   W∈Y ≡ TT
subset(X,Y)≡TT, W∈Y≡FF  ⊢   W∈X ≡ FF
isset(X) ≡ TT      ⊢   subset(X,X) ≡ TT
isset(X)≡TT,isset(Y)≡TT, ∀W. W∈X::W∈Y≡TT  ⊢   subset(X,Y)≡TT
subset(X,Y)≡TT     ⊢   ∀W. W∈X :: W∈Y ≡ TT
subset(X,NS)≡TT    ⊢   X ≡ NS
subset(W,X)≡TT, subset(X,Y)≡TT            ⊢   subset(W,Y) ≡ TT
```

APPENDIX 12 (continued).

d) The usual union operation - '∪' .

```
                  ⊢  ∀X . X∪UU ≡ UU
                  ⊢  ∀X . UU∪X ≡ UU
isset(X) ≡ FF     ⊢  ∀Y . X∪Y ≡ UU
isset(Y) ≡ FF     ⊢  ∀X . X∪Y ≡ UU
∂(X∪Y) ≡ TT       ⊢  isset(X) ≡ TT
∂(X∪Y) ≡ TT       ⊢  isset(Y) ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT  ⊢  isset(X∪Y) ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT , X∪Y ≡ UU  ⊢  TT ≡ FF
W∈X ≡ TT, isset(Y) ≡ TT  ⊢  W∈(X∪Y) ≡ TT
W∈Y ≡ TT, isset(X) ≡ TT  ⊢  W∈(X∪Y) ≡ TT
W∈X ≡ FF, W∈Y ≡ FF       ⊢  W∈(X∪Y) ≡ FF
W∈(X∪Y) ≡ FF   ⊢  W∈X ≡ FF
W∈(X∪Y) ≡ FF   ⊢  W∈Y ≡ FF
isset(X) ≡ TT, isset(Y) ≡ TT  ⊢  subset(X,X∪Y) ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT  ⊢  subset(Y,X∪Y) ≡ TT
isset(X) ≡ TT  ⊢  X∪NS ≡ X
isset(X) ≡ TT  ⊢  NS∪X ≡ X
isset(X) ≡ TT  ⊢  X∪X ≡ X
subset(X,Y)≡TT ⊢  X∪Y ≡ Y
                  ⊢  X∪Y ≡ Y∪X
                  ⊢  ∀X Y Z . (X∪Y)∪Z ≡ X∪(Y∪Z)
```

e) The set subtraction (\ ) operation.

```
                  ⊢  ∀X . X\UU ≡ UU
                  ⊢  ∀X . UU\X ≡ UU
isset(X) ≡ FF     ⊢  ∀Y . X\Y ≡ UU
isset(Y) ≡ FF     ⊢  ∀X . X\Y ≡ UU
∂(X\Y) ≡ TT       ⊢  isset(X) ≡ TT
∂(X\Y) ≡ TT       ⊢  isset(Y) ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT      ⊢  isset(X\Y) ≡ TT
isset(X)≡TT, isset(Y)≡TT, X\Y≡UU  ⊢  TT ≡ FF
W∈X ≡ FF, isset(Y) ≡ TT           ⊢  W∈(X\Y) ≡ FF
W∈Y ≡ TT, isset(X) ≡ TT           ⊢  W∈(X\Y) ≡ FF
W∈X ≡ TT, W∈Y ≡ FF                ⊢  W∈(X\Y) ≡ TT
W∈(X\Y) ≡ TT   ⊢  W∈X ≡ TT
W∈(X\Y) ≡ TT   ⊢  W∈Y ≡ FF
isset(X) ≡ TT, isset(Y) ≡ TT      ⊢  subset(X\Y,X) ≡ TT
isset(X) ≡ TT  ⊢  X\X ≡ NS
isset(X) ≡ TT  ⊢  X\NS ≡ X
isset(X) ≡ TT  ⊢  NS\X ≡ NS
```

APPENDIX 12 (continued).

f) Properties of usual intersection operation - '∩' .

```
                    ⊢  ∀X . X∩UU ≡ UU
                    ⊢  ∀X . UU∩X ≡ UU
isset(X) ≡ FF       ⊢  ∀Y . X∩Y ≡ UU
isset(Y) ≡ FF       ⊢  ∀X . X∩Y ≡ UU
∂(X∩Y) ≡ TT         ⊢  isset(X) ≡ TT
∂(X∩Y) ≡ TT         ⊢  isset(Y) ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT          ⊢  isset(X∩Y) ≡ TT
isset(X)≡TT, isset(Y)≡TT, X∩Y≡UU      ⊢  TT ≡ FF
W∈X ≡ FF , isset(Y) ≡ TT              ⊢  W∈(X∩Y) ≡ FF
W∈Y ≡ FF , isset(X) ≡ TT              ⊢  W∈(X∩Y) ≡ FF
W∈X ≡ TT , W∈Y ≡ TT                   ⊢  W∈(X∩Y) ≡ TT
W∈(X∩Y) ≡ TT        ⊢  W∈X ≡ TT
W∈(X∩Y) ≡ TT        ⊢  W∈Y ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT          ⊢  subset(X∩Y,X) ≡ TT
isset(X) ≡ TT, isset(Y) ≡ TT          ⊢  subset(X∩Y,Y) ≡ TT
isset(X) ≡ TT       ⊢  X∩NS ≡ NS
isset(X) ≡ TT       ⊢  NS∩X ≡ NS
isset(X) ≡ TT       ⊢  X∩X ≡ X
                    ⊢  X∩Y ≡ Y∩X
                    ⊢  ∀X Y Z . (X∩Y)∩Z ≡ X∩(Y∩Z)
```

g) The 'select' function.

```
                    ⊢  select(UU) ≡ UU
                    ⊢  select(NS) ≡ UU
isset(X) ≡ FF       ⊢  select(X) ≡ UU
∂(select(X))≡TT     ⊢  isset(X) ≡ TT
∂(select(X))≡TT     ⊢  (X≡NS) ≡ FF
isset(X) ≡ TT, (X≡NS) ≡ FF  ⊢  ∂(select(X)) ≡ TT
isset(X) ≡ TT, (X≡NS) ≡ FF  ⊢  select(X)∈X ≡ TT
```

h) The 'singtn' function.

```
                    ⊢  singtn(UU) ≡ UU
∂(X) ≡ TT           ⊢  isset(singtn(X)) ≡ TT
∂(singtn(X))≡TT     ⊢  ∂(X) ≡ TT
∂(X) ≡ TT           ⊢  X∈singtn(X) ≡ TT
X∈singtn(Y)≡TT      ⊢  X ≡ Y
∂(X) ≡ TT           ⊢  (singtn(X)≡NS) ≡ FF
∂(X) ≡ TT           ⊢  select(singtn(X)) ≡ X
```